



SOFTWARE TESTING

Prof. Meenakshi D'souza

Computer Science and Engineering

IIIT Bangalore-IISc



INDEX

<u>S.No.</u>	<u>Topic</u>	<u>Page No.</u>
<i>WEEK- 1</i>		
1	Motivation	01
2	Terminologies	14
3	Testing based on Models and Criteria	31
4	Automation JUnit as an example	50
<i>WEEK- 2</i>		
5	Basics of Graphs: As used in testing	70
6	Structural Graph Coverage Criteria	91
7	Elementary Graph Algorithms	115
8	Elementary Graph Algorithms- 2	134
9	Algorithms: Structural Graph Coverage Criteria	150
<i>WEEK- 3</i>		
10	Structural Coverage Criteria	171
11	Data Flow Graphs	183
12	Algorithms: Data Flow Graph Coverage Criteria	204
13	Graph Coverage Criteria: Applied to Test Code	222
14	Testing Source Code: Classical Coverage Criteria	244
<i>WEEK – 4</i>		
15	Data Flow Graph Coverage Criteria : Applied to Test Code	261
16	Software Design and Integration Testing	281
17	Design Integration Testing and Graph Coverage	298
18	Specification Testing and Graph Coverage	322
19	Graph Coverage and Finite state Machines	339

WEEK – 5

20	Graph Coverage Criteria	358
21	Basics Needed for Software Testing	374
22	Logic: Coverage Criteria	395
23	Coverage Criteria, Contd.	419
24	Logic Coverage Criteria (Contd.)	437

WEEK – 6

25	Logic Coverage Criteria: Applied to Test Code_1	456
26	Logic Coverage Criteria: Applied to Test Code_2	474
27	Logic Coverage Criteria: Issues in Applying to Test Code	492
28	Logic Coverage Criteria: Applied to Test Specifications	507
29	Logic Coverage Criteria: Applied to Finite State Machines	523

WEEK – 7

30	Assignment Solving	535
31	Functional Testing	548
32	Input Space Partitioning	569
33	Input Space Partitioning: Coverage Criteria	590
34	Input Space Partitioning Coverage Criteria: Example	605

WEEK – 8

35	Syntax-Based Testing	618
36	Mutation Testing	635
37	Mutation Testing for Programs	648
38	Mutation Testing: Mutation Operators for Source Code	667
39	Mutation Testing Vs. Graphs and Logic Based Testing	684

WEEK – 9

40	Mutation testing	700
41	Mutation Testing Mutation for integration	721
42	Mutation testing Grammars and inputs	745
43	Software Testing Course Summary after week 9	764

WEEK – 10

44	Testing of web Applications and Web Services	776
45	Testing of web Applications and Web Services	794
46	Testing of web Applications and Web Services	811
47	Testing of Object-Oriented Applications	834
48	Testing of Object-Oriented Applications	850

WEEK – 11

49	Symbolic Testing	863
50	Symbolic Testing- 2	882
51	DART: Directed Automated Random Testing-1	898
52	DART: Directed Automated Random Testing-2	916
53	DART: Directed Automated Random Testing-3	934

WEEK – 12

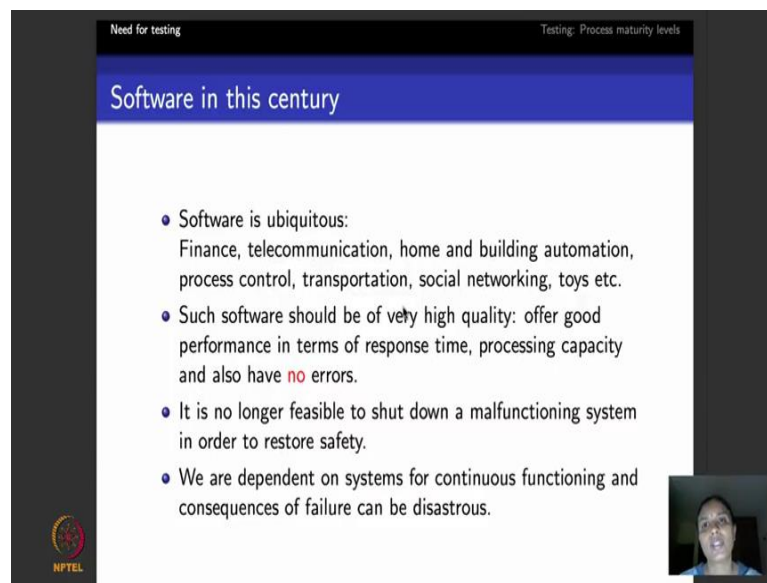
54	Testing of Object-Oriented Applications	949
55	Testing of Mobile Applications	965
56	Non-Functional System Testing	985
57	Regression Testing	1005
58	Assignment: Week 11 Solving	1019
59	Software Testing: Summary at the End of the Course	1032

Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 01
Software Testing: Motivation

Hello everyone my name is Meenakshi. And this is a first lecture that I will be doing as a part of the course on software testing that I am currently offering for NPTEL. So, what are we going to do today? Basically I will talk to you about the kinds of software that we encounter in the present world, and what do expensive errors look like, and how to avoid these expensive errors by using testing. So, it is basically motivation module where we deal with how important testing is.

(Refer Slide Time: 00:45)



The slide is titled "Software in this century" and is part of a presentation on "Need for testing" and "Testing: Process maturity levels". It features a list of four bullet points:

- Software is ubiquitous:
Finance, telecommunication, home and building automation, process control, transportation, social networking, toys etc.
- Such software should be of very high quality: offer good performance in terms of response time, processing capacity and also have **no** errors.
- It is no longer feasible to shut down a malfunctioning system in order to restore safety.
- We are dependent on systems for continuous functioning and consequences of failure can be disastrous.

The NPTEL logo is visible in the bottom left corner, and a small video inset of the professor is in the bottom right corner.

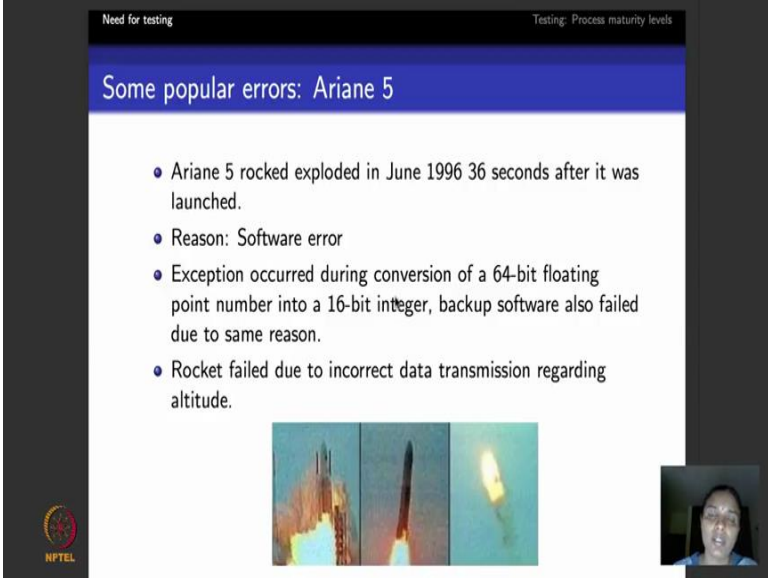
So, what a software in the current century look like? If you look all around you, it is not too difficult to understand that there is software.

So, we use Paytm, we use banks online, we go to an ATM machine there is software, I make a call to my friend or my parents using my mobile phones, there is software. Software that controls the heating, ventilation then air conditioning in our homes and offices, there is software that tells you how electricity flows through a grids, how water flows through a network. Software that run in manufacturing industries, the software that help us to drive a car, autopilot and other kinds of software that help you to fly our

planes, when the software that controls real network, the software even in toys that children use right.

So, what is our expectation about such software? We all expect the software to be error free, not only that we want the software to respond fast right. The second I insert my card into the ATM machine and what the welcome screen the next second before I had link my id, I cannot be it long enough. And the other important thing to notice that if you consider a software like the autopilot of a plane, let us say there is an error in the software you cannot say that I will shut it down mid flight, I will rectify the bug and I will restart to be able to run right. It is no longer feasible to be able to shut down system because there is an error in the software. We want the system to be able to continue to done in the autopilot case it needs to be able to continue to fly the aircraft safely and help us to land safely.

(Refer Slide Time: 02:22)



The slide is titled "Some popular errors: Ariane 5" and is part of a presentation on "Need for testing" and "Testing: Process maturity levels". It lists four bullet points:

- Ariane 5 rocket exploded in June 1996 36 seconds after it was launched.
- Reason: Software error
- Exception occurred during conversion of a 64-bit floating point number into a 16-bit integer, backup software also failed due to same reason.
- Rocket failed due to incorrect data transmission regarding altitude.

Below the text is a video frame showing the Ariane 5 rocket launch. The NPTEL logo is in the bottom left corner, and a small video inset of the presenter is in the bottom right corner.

So, I would like to talk to you about some popular errors that have occurred in the past. This was the error is the error is the error in European space agency rocket called Ariane 5, it occurred exactly 21 years ago and the important thing to note that this error is because of a software bug. So, there was this rocket called Ariane 5 which was being controlled by software that is launched by European space agency.

So, this software had the following error. So, it was trying to squeeze in data corresponding to a 64-bit floating point number into a memory space that is allotted for a

16-bit integer. So obviously it is not going to be able to succeed in doing that. And because such rockets have safety critical systems they always have backup software, but in this case the problem was the backup software also had exactly the same error. So, this resulted in transmission of incorrect altitude data to the aircraft and this rocket Ariane 5, went and plunged into the Atlantic Ocean within 36 seconds after it was launched. So, that is about 15 years of total effort and you can imagine that millions of Euros that would have been lost.

(Refer Slide Time: 03:33)

The slide is titled "Some popular errors: Therac 25". It contains two bullet points: "Six patients died due to an overdose of radiation caused from Therac 25, a medical linear accelerator that is used to treat tumors." and "The main cause of error was a race condition caused by wrong sequence of commands caused by the software operating the accelerator." Below the text is a photograph of the Therac 25 machine. The slide also features a small video inset of a person in the bottom right corner and an NPTEL logo in the bottom left corner.

Need for testing

Testing: Process maturity levels

Some popular errors: Therac 25

- Six patients died due to an overdose of radiation caused from Therac 25, a medical linear accelerator that is used to treat tumors.
- The main cause of error was a race condition caused by wrong sequence of commands caused by the software operating the accelerator.

NPTEL

So, the next example that I would like to discuss about is another unfortunate example that happened, again because of a software error. So, in this case 6 patients lost their life due to buggy software in a machine that gives radiation therapy to cancer patients. So, there was a race condition error in this software. So, this software ended up calculating more dosage of radiation then what was needed and unfortunately these patients lost their life because of an overdose of radiation.

(Refer Slide Time: 04:05)

Need for testing

Testing: Process maturity levels

Some popular errors: Intel Pentium Bug

- Intel lost an estimated \$475 million due to a defective pentium chip.
- The chip made mistakes while dividing floating point numbers within a certain range.
- For example, $3145727 \times 4195835 / 3145727$ should return 4195835. A flawed Pentium will return 4195579!
- Intel had to replace most of 3 to 5 million defective chips in circulation.

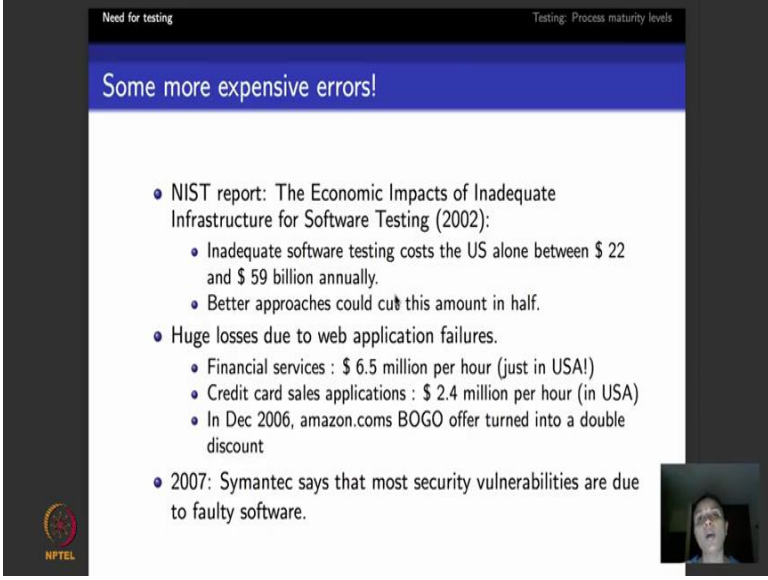
NPTEL

Small video inset showing a person speaking.

So, the next kind of example that I would like to discuss with you it is software that is expensive that costssomebody a lot. You all would have heard of this Intel Pentium processors right. So, there was this particular P4 Pentium processor, the mathematician in the US, when he was trying to do research towards its prime numbers, he found that this Pentium 4 processor was doing floating point division wrongly. So, when it was announced an Intel investigator did realize that all the P4 processors that it had released to the market had the same error. So, the only solution left was to be able to recall all these processors and that cost Intel a lot of money.

So, this does not bring us to the end of expensive errors. So, if you Google you know you can talk, you can find Toyota breaks, crashes all kinds of crashes happened.

(Refer Slide Time: 04:56)



The slide is titled "Some more expensive errors!" and is part of a presentation on software testing. It features a blue header bar with the title. The main content is a list of bullet points. The first bullet point is a NIST report from 2002 about the economic impacts of inadequate infrastructure for software testing. The second bullet point is about huge losses due to web application failures, with sub-bullets for financial services, credit card sales, and a specific Amazon.com incident in December 2006. The third bullet point is from 2007, stating that most security vulnerabilities are due to faulty software. The slide also has a small video inset of a speaker in the bottom right corner and an NPTEL logo in the bottom left corner.

Need for testing

Testing: Process maturity levels

Some more expensive errors!

- NIST report: The Economic Impacts of Inadequate Infrastructure for Software Testing (2002):
 - Inadequate software testing costs the US alone between \$ 22 and \$ 59 billion annually.
 - Better approaches could cut this amount in half.
- Huge losses due to web application failures.
 - Financial services : \$ 6.5 million per hour (just in USA!)
 - Credit card sales applications : \$ 2.4 million per hour (in USA)
 - In Dec 2006, amazon.coms BOGO offer turned into a double discount
- 2007: Symantec says that most security vulnerabilities are due to faulty software.

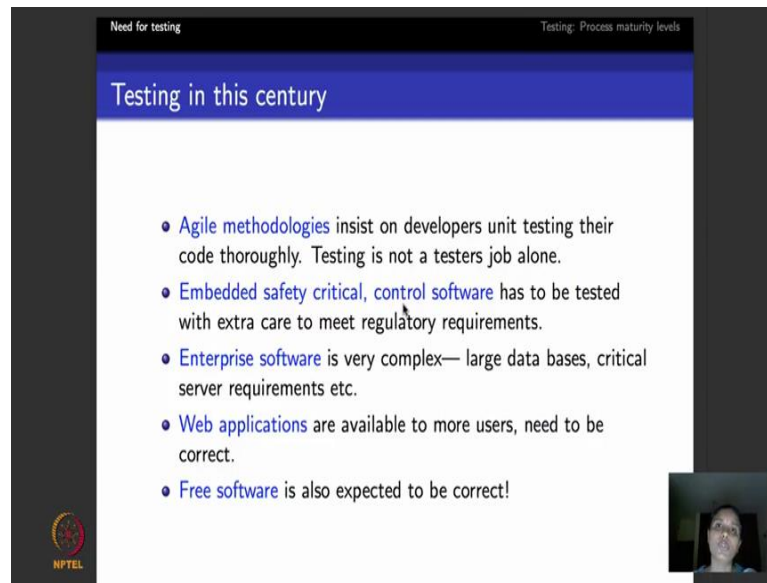
NPTEL

So, here are some few reports that talk about how expensive errors can be and what is the cost of inadequate testing. National institute standards and technology in the year 2002 released a report which basically discussed the impact of software testing in US.

So, it says that inadequate software testing causes the US economy anywhere between 22 and 59 billion every year. Not only that the report goes ahead and says that if there are better approaches that can be found to do software testing, then you could bring down the amount of these losses to almost half of what it is. The other popular categories of losses are due to web applications which we use all the time. So, one particular thing that I would like to discuss with you now happened almost 10 years ago. Amazon had this online discount sale, and because of a software error what happened was that it ended up giving double the amount of discount and by the time Amazon realized it, it is too late that are already lost lot of money.

The next fact that I would like to drought your attention is a report that Symantec, the security organization published in 2007. So, it says that security related vulnerabilities that occur in a financial transactions, in our online wallets and so on mainly occur no because of cryptographic errors, they occur because of software errors.

(Refer Slide Time: 06:21)



The slide is titled "Testing in this century" and is part of a presentation on "Need for testing" and "Testing: Process maturity levels". It lists five types of software that require testing:

- Agile methodologies insist on developers unit testing their code thoroughly. Testing is not a testers job alone.
- Embedded safety critical, control software has to be tested with extra care to meet regulatory requirements.
- Enterprise software is very complex— large data bases, critical server requirements etc.
- Web applications are available to more users, need to be correct.
- Free software is also expected to be correct!

The slide also features the NPTEL logo in the bottom left corner and a small video feed of the presenter in the bottom right corner.

So, we know expensive errors and software can be. So, let us try to look at the various kinds of software and typically how they are developed. You might have heard this buzzword called agile methodologies right. So, agile methodologies basically insist that people who develop the code, that is the developers, also have to unit test a code. Typically, developers do not have any great knowledge on testing, but agile methodology believes that the developer himself or herself the best person to identify the error in the code. So, that puts a lot of pressure on developers to be able to know testing well.

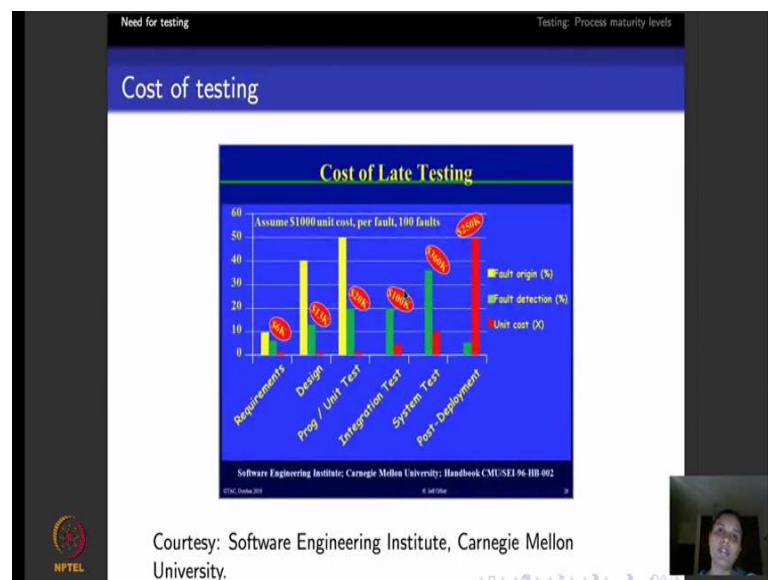
Now, let us look at the various kinds of software that we will deal with. If you consider a software that typically runs in your cars that helps you to do breaking automatically, or does a cruise control or a software that runs in your aeroplanes, such software is what is called embedded control software. So obviously, such software runs in what are called safety critical systems where a failure can be catastrophic. So, safety critical software have to meet regulatory standards where, by which regulatory authorities decide if a software has been tested enough and it is fit for release. So, this just means that almost double the effort goes into testing the software.

Other kind of software is what is called enterprise software. So, what would be a typical example of an enterprise software let us see a software that run city bank right or a software that managers are Indian railways ticketing system right. What do such software

have to deal with, the basically the first complex thing is that they deal with fairly huge data basis that have a lot of data and the software is critically dependent on the server and backup server running all the time right.

The third popular category of software is what is called web applications. Things like social networking sites, amazon online and so on and so forth. We all know how important it is for these software to be correct. Finally, I would like to end this slide with the point what free software suppose you pick up a piece of software from the internet for free. Just because it is available for free you are not willing to accept the fact this software could have errors. So, paradoxically we expect free software also to be error free.

(Refer Slide Time: 08:41)



This is a very important slide. So, this is a part of study that was done at Carnegie Mellon university a few years ago. So, this slide discusses about how expensive testing can get as we go down software development.

So, if you see a typical software development initially you write requirements and then you do design right. And then your unit test your software and then put together the modules, that you have tested and do what is called integration testing then you put the software with the system or hardware that it is supposed to run on and in your system testing and finally, release the software.

So, suppose there was an error in the requirements, and it was found that and there then what the slide tells you is that gives you the cost - the cost is fairly low right, but suppose there was an error that was found in requirements or design, and it went all the way through testing, integration testing, system testing, did not get detected at all. The software called released and the error was found there. Then what this slide tells you is that the cost fixing that can be very high, it is not only the cost fixing the error the consequences of that error as we saw through the examples can also be really bad right.

(Refer Slide Time: 09:49)

Need for testing

Testing: Process maturity levels

Testing: Facts and Myths

- Fact: Testing can be used to find errors in software, cannot be used to show that a software is correct.
- Fact: Testing cannot be replaced by software reviews, inspections, quality audits etc.
- Fact: Testing cannot be fully automated, needs human intervention.
- Myth: It is wrong to say that "My code is correct and doesn't need to be tested".

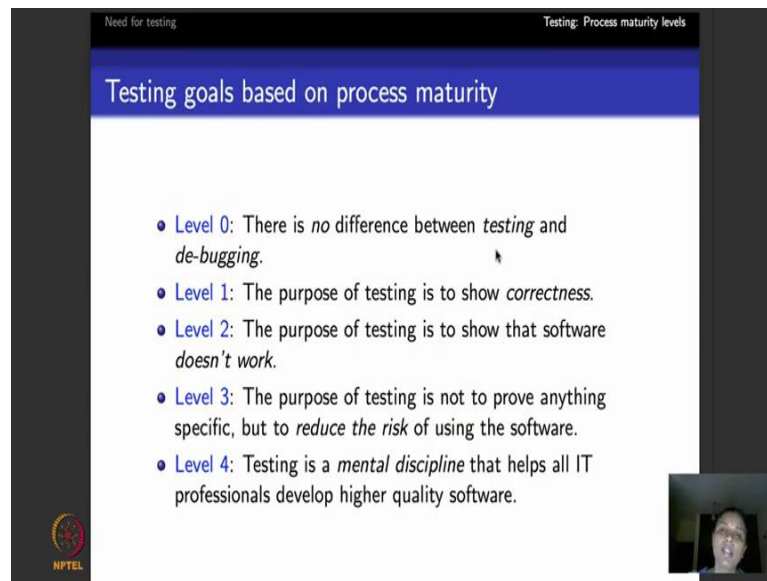
NPTEL

So, what are the facts and myths about testing? There is a popular saying by Edgar Dijkstra who said that never think that you can use testing and say that I have proved my software to be correct. It is wrong to say the testing prove software to be correct. The main goal of testing is to be able to find errors in the software right. Using testing I can never say that I have tested my software and it is fully correct, it is a wrong statement to make. The other wrong statement to make is that a developer typically thinks that you know “ have written the code, I have debugged it well and my quality auditor my SQA in my company has inspected the code. So, there is no need to test it”. That typically will not work; it still needs to be tested.

Another popular thought that people have especially these people who sell software testing tools is to be able to say that you just download and install my tool it can do magic; it will do every kind of testing you can think of. That is not true.

The main goal of testing tools is to be able to help you execute the test cases and record the result. Now to be able to design test cases, you will have to be able to design cases to find errors effectively. Typical Pareto principle applies here 80 percent of the errors come in 20 percent of the code or design. So, test case design which is what the course is about needs a human to be able to do it right.

(Refer Slide Time: 11:18)



The slide is titled "Testing goals based on process maturity" and is part of a presentation on "Need for testing" and "Testing: Process maturity levels". It lists five levels of testing maturity:

- **Level 0:** There is *no* difference between *testing* and *de-bugging*.
- **Level 1:** The purpose of testing is to show *correctness*.
- **Level 2:** The purpose of testing is to show that software *doesn't work*.
- **Level 3:** The purpose of testing is not to prove anything specific, but to *reduce the risk* of using the software.
- **Level 4:** Testing is a *mental discipline* that helps all IT professionals develop higher quality software.

The slide also features an NPTEL logo in the bottom left corner and a small video inset of a person in the bottom right corner.

So, I would like to end this module by discussing about certain process maturity levels in testing. Why is it important to look at process maturity levels? It is important because it tells you what role testing plays in the software development that you do. Broadly there are 5 levels beginning from 0 and going all the way till 4.

(Refer Slide Time: 11:42)

Need for testing

Testing: Process maturity levels

Testing process: Level 0 thinking

- Testing is the same as debugging.
- Does not distinguish between incorrect behavior and mistakes in the program.
- Does not help develop software that is reliable or safe.

NPTEL

A small video inset in the bottom right corner shows a man speaking.

So, what does level 0 tell you? Level 0 tells you that there is basically nothing called testing. Developers write their code, they debug their code they ensure that it is correct and that is it, they go ahead and release it right. So, it is clear that this does not really help to develop software that is considered to be fully safe and reliable.

(Refer Slide Time: 12:01)

Need for testing

Testing: Process maturity levels

Testing process: Level 1 thinking

- Purpose is to show correctness. Correctness is impossible to achieve.
- What do we know if no failures? Good software or bad tests?
- Test engineers have no:
 - Strict goal
 - Real stopping rule
 - Formal test technique

NPTEL

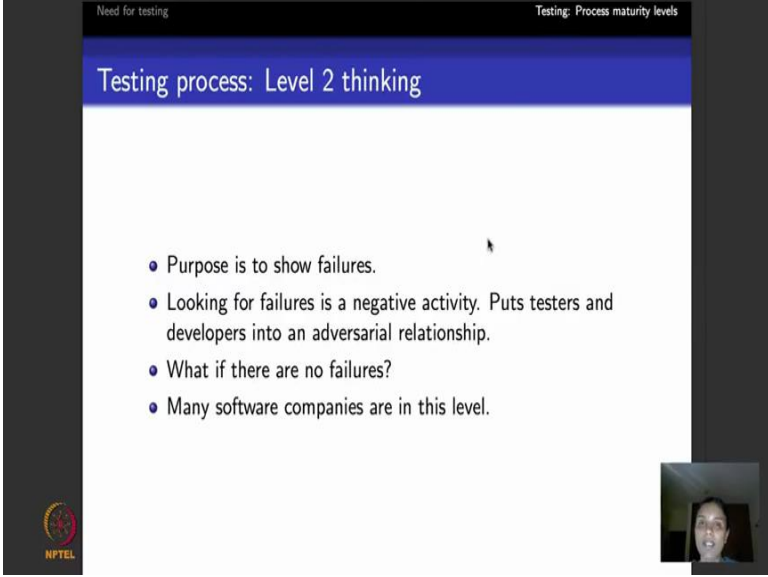
A small video inset in the bottom right corner shows a man speaking.

The next comes level one thinking. Level one thinking a developer thinks that he or she has written a piece of code and their goal to test is to be able to show that the code is correct. As we saw little while ago I clearly cannot use testing to show that a piece of

software it is correct right. So, let us see a particular piece of code has an integer variable. To be able to exhaustively tested on the 32 bit processor I need to be able to give every value which is $2^{32} - 1$ to $2^{32} + 1$, and even when an extra fast PC this is going to be able to take several years to do. So, unless I test it for every value I cannot say that the testing is correct.

So, here there is nothing like test engineers and they even if there are they do not have goals and they just show that the software is not failed, but the underlined listen it is not failed because it is correct or it is not failed because you have designed the test cases wrongly, that is never clear.

(Refer Slide Time: 13:00)

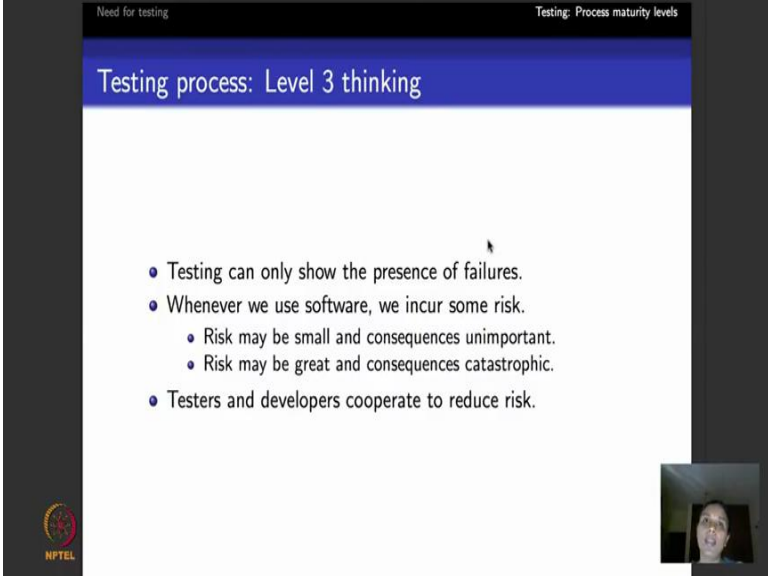


The slide is titled "Testing process: Level 2 thinking" and is part of a presentation on "Testing: Process maturity levels". It lists four bullet points: "Purpose is to show failures.", "Looking for failures is a negative activity. Puts testers and developers into an adversarial relationship.", "What if there are no failures?", and "Many software companies are in this level." A small video inset in the bottom right corner shows a person speaking. The NPTEL logo is in the bottom left corner.

- Purpose is to show failures.
- Looking for failures is a negative activity. Puts testers and developers into an adversarial relationship.
- What if there are no failures?
- Many software companies are in this level.

The next level is level 2 thinking when you begin to believe that the goal of testing is to be able to identify failures or errors in the software. This is the beginning of positively using testing, but in organizations that are typically at this level there is a lot of tiff between developers and testers, they belong to different teams and then one does not want to help the other, and there is confusion even though the goal of testing is fully realized. We move on to level 3 where they not only realize that the goal of testing is to find errors, but they also work together and say that we will not only find errors, we will make sure that we reduce errors to the extent possible in the software.

(Refer Slide Time: 13:28)



Need for testing

Testing: Process maturity levels

Testing process: Level 3 thinking

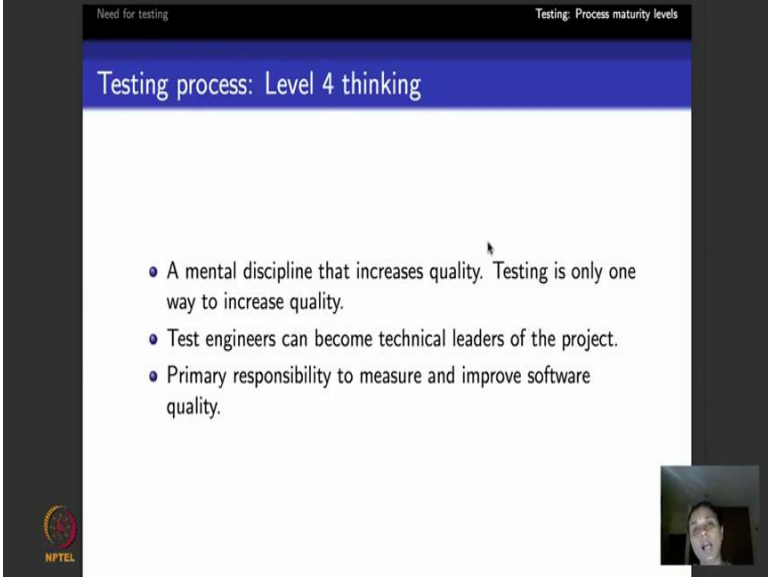
- Testing can only show the presence of failures.
- Whenever we use software, we incur some risk.
 - Risk may be small and consequences unimportant.
 - Risk may be great and consequences catastrophic.
- Testers and developers cooperate to reduce risk.

NPTel

A small video inset in the bottom right corner shows a man speaking.

And that is also conscious understanding that when I release the software there is a bit of risk involved. The risk could be high or the risk could be low and my goal is to make the risk as low as possible.

(Refer Slide Time: 13:55)



Need for testing

Testing: Process maturity levels

Testing process: Level 4 thinking

- A mental discipline that increases quality. Testing is only one way to increase quality.
- Test engineers can become technical leaders of the project.
- Primary responsibility to measure and improve software quality.

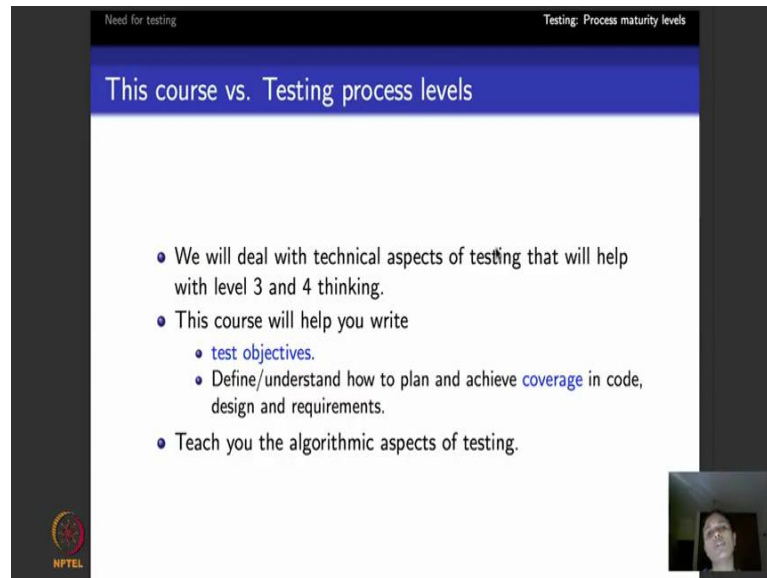
NPTel

A small video inset in the bottom right corner shows the same man speaking.

Level 4 thinking is where large organizations strive to be in, here testing becomes a mental discipline. So, there is positive thought and efforts in the organization level to make sure that testing teams' efforts are taken into consideration to develop software that is as error free as possible. An organization like Microsoft which try to achieve

excellence in level 4 thinking and this is what is the desired level that we would like to achieve in testing.

(Refer Slide Time: 14:28)



The slide is titled "This course vs. Testing process levels" and is part of a presentation on testing. It features a blue header bar with the title. The main content area is white with a list of bullet points. The top left corner of the slide has the text "Need for testing" and the top right corner has "Testing: Process maturity levels". The NPTEL logo is in the bottom left corner. A small video inset of the speaker is in the bottom right corner.

- We will deal with technical aspects of testing that will help with level 3 and 4 thinking.
- This course will help you write
 - test objectives.
 - Define/understand how to plan and achieve coverage in code, design and requirements.
- Teach you the algorithmic aspects of testing.

Now, what is this course got to do with all the levels and terminologies that we saw till now? So, this course will help you to think that I want to be able to do testing at levels 2 3 or 4, which means what that the goal of my testing is to be able to find errors. So, to find errors, I have to first define what are my testing objectives in technical terms, and I have to be able to figure out how to effectively design test cases to be able meet or cover these test objectives.

So, in the course we will look at algorithms and techniques that will help you to formulate test objectives and design test cases that will help you to meet these objectives. So, the next module that we will be seeing, we will introduce the various terminologies that exist in testing and also clarify about what we would use is a part of this course.

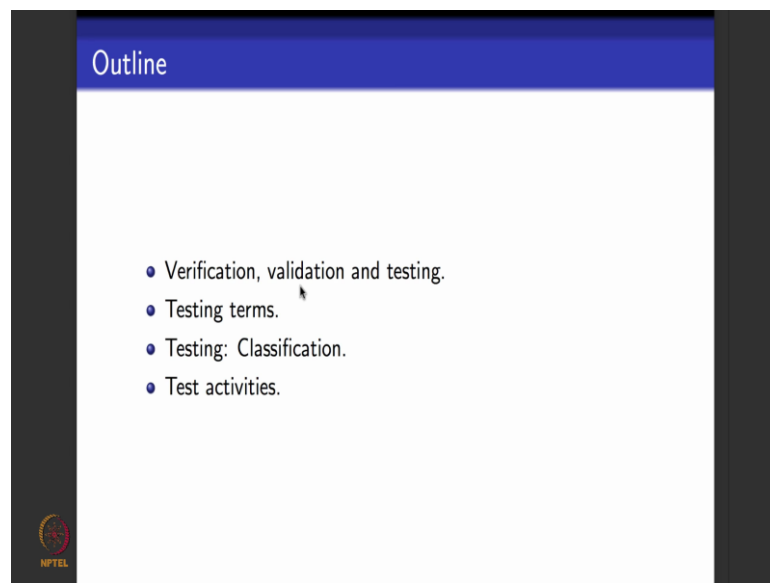
Thank you.

Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 02
Software Testing: Terminologies

Hello everyone. So, this is a first module of the first week. So, in this module what I will be doing is to introduce you to the various terminologies that exist in the area of software testing. What we would be using in this course - clarify the basic terms and tell you what are the various types of testing, the various methods in testing. We will also look at various activities in testing and see what this course will deal with. Here is an outline of the course.

(Refer Slide Time: 00:39)

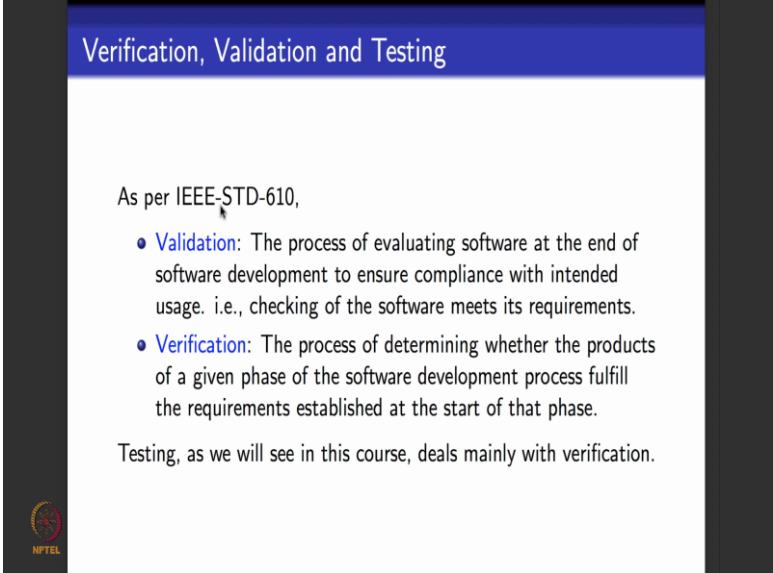


So, to begin with we clarify on these terms which you might have heard about verification, validation and testing and I will tell you what we will deal with in this course. We will also look at a whole set of related areas briefly and clarify what this course will cover. There are several terms and terminologies related to testing. What is the test case? What is an error? What is a fault? What is a failure? So, I will introduce you to all these testing terms and we will see what they mean.

And then in testing there are several methods of testing, several types of testing, several phases of testing. You might have heard terms like white box, black box, functional

testing, usability testing, performance testing. So, we will see what these various terms are in the classification of testing. I will end this module by introducing you to various testing activities and tell you what our course is going to focus on.

(Refer Slide Time: 01:36)




Verification, Validation and Testing

As per IEEE-STD-610,

- **Validation:** The process of evaluating software at the end of software development to ensure compliance with intended usage. i.e., checking of the software meets its requirements.
- **Verification:** The process of determining whether the products of a given phase of the software development process fulfill the requirements established at the start of that phase.

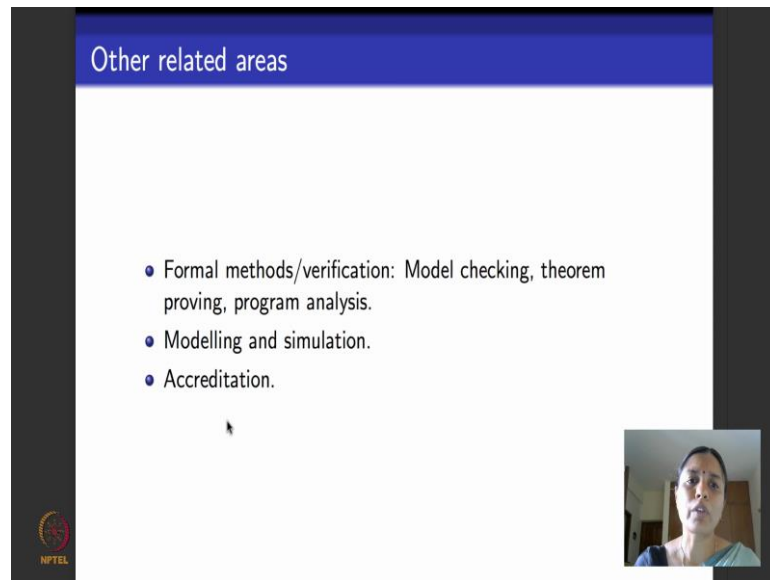
Testing, as we will see in this course, deals mainly with verification.



IEEE maintains a standard glossary of software engineering terms in this particular standard called STD-610. So, as per that standard what is verification and what is validation? Validation is something that you do at the end of software development or system development. You may have your own software or system developed ready in place and you want to be able to check if the software or system meets all its requirements when you do that at the end of the development then that is what is called validation. Verification deals with what you test or verify while developing software. So, while developing software you might go through various phases, phase where you define requirements, phase where you do design and architecture. Phase where write code and phase where you test. At each stage you are dealing with a set of software artifacts.

So, the kind of verification that you do where in you check whether a particular software artifact needs the requirements that were established for that particular set of artifacts then you do what is called verification. Testing as we will see in this course mainly deals with verification. There are several other related areas that you might have heard about.

(Refer Slide Time: 02:46)



One of the most popular or my favorite ones is that of formal methods, formal verification. There are 3 broad areas that people work on within formal verification namely model checking, theorem proving, program analysis. Testing as we will see in this course, we will not deal with model checking or theorem proving or program analysis. Towards the end of the course we will see symbolic execution. A lot of symbolic execution is used in program analysis, but we will see it from the point of view of testing.

In fact, there are several NPTEL courses available in each of these areas and if interested you are more than welcome to go and see them.

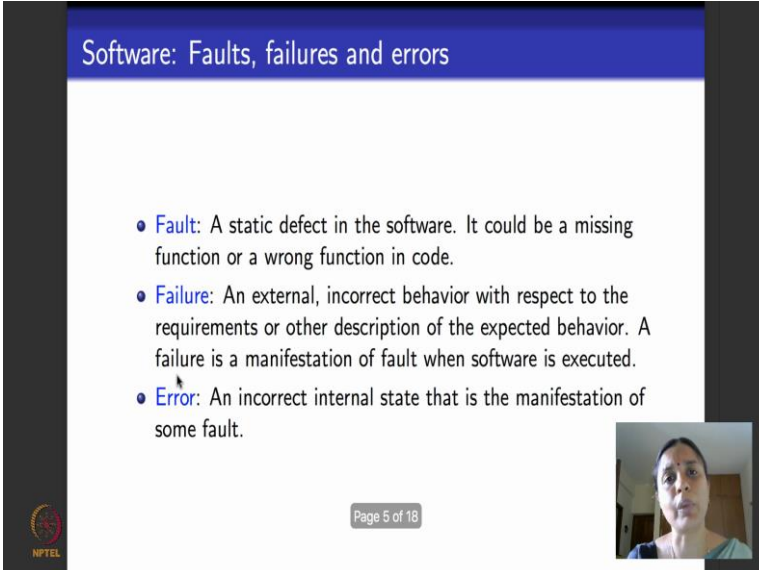
A related area where people also do verification is what is called modeling and simulation. Let us see if you have a piece of hardware design or a system design or a software design. For hardware design popular languages that you could use a VHDL very low, so you have your design and then you model your design in this particular language and you run simulations to see if the design model that you have done correctly does the design as per its requirements. So, you have a modeling language, which is usually proprietary or open source you modeled using that modeling language and run several different kinds of simulations to see if the design model needs its requirements.

So, that area is modeling in simulation and the testing that we would see in this course we will not deal with that also. Another related area is what is called accreditation. Now

what is accreditation? Accreditation deals with software that is safety critical and needs to be certified. Take for example, software that runs a plane, right, like typical autopilot software, any company you cannot say that I have written in autopilot software as per the normal requirements of autopilot software. So, here it is, take and run right. So, these autopilot softwares being safety critical because they cannot afford to fail they have to be audited and accredited by responsible authorities like FAA in the US, DGCA in India and they demand lot of additional testing there is a separate process of accreditation for which there is testing needs to be done.

But accreditation as it is, we won't look at, in this course what we will look at is the kind of testing that people do towards accreditation and the algorithm behind that kind of testing.

(Refer Slide Time: 05:02)



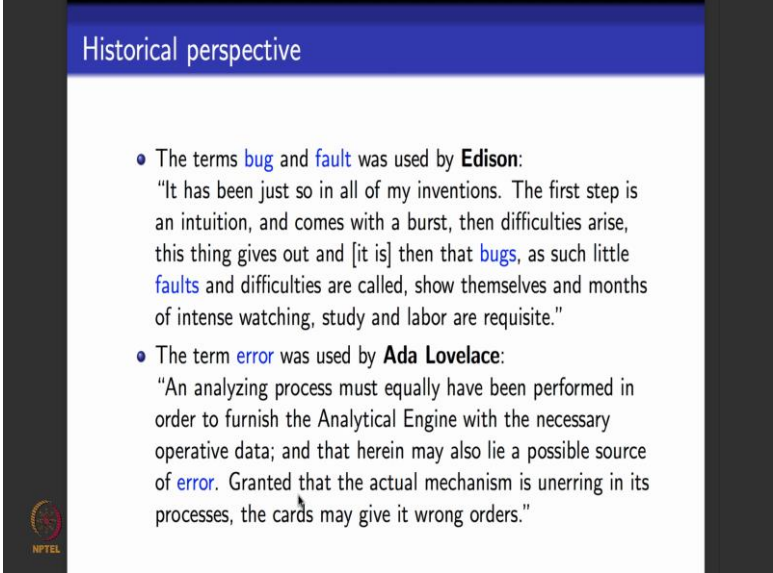
The slide is titled "Software: Faults, failures and errors" in a blue header. It contains three bullet points defining the terms. In the bottom right corner, there is a small video inset showing a woman speaking. The NPTEL logo is in the bottom left, and "Page 5 of 18" is in the bottom center.

- **Fault:** A static defect in the software. It could be a missing function or a wrong function in code.
- **Failure:** An external, incorrect behavior with respect to the requirements or other description of the expected behavior. A failure is a manifestation of fault when software is executed.
- **Error:** An incorrect internal state that is the manifestation of some fault.

We will move on to looking at various terms in testing, you might have heard of lot of these terms faults, failures, errors ... what do they mean? So, what is a fault? Fault is considered to be a static defect that occurs in software, a static defect in the sense the defect that occurs as a part of the software, not a defect that occurs when the software is executed. So, this defect could be a function that was wrongly written, function that was not written at all, it could be any of these, right and then when software with the static defect or fault is executed, there is a misbehavior or a wrong behavior on software that is observed, right. This wrong, observable behavior is what is called failure. When the

software has a fault to execute around the software you observe a failure, then the software is supposed to enter an incorrect state. The incorrect state in which software enters, the software that has a fault enters and a failure is manifested is what is called the error in the software.

(Refer Slide Time: 06:07)

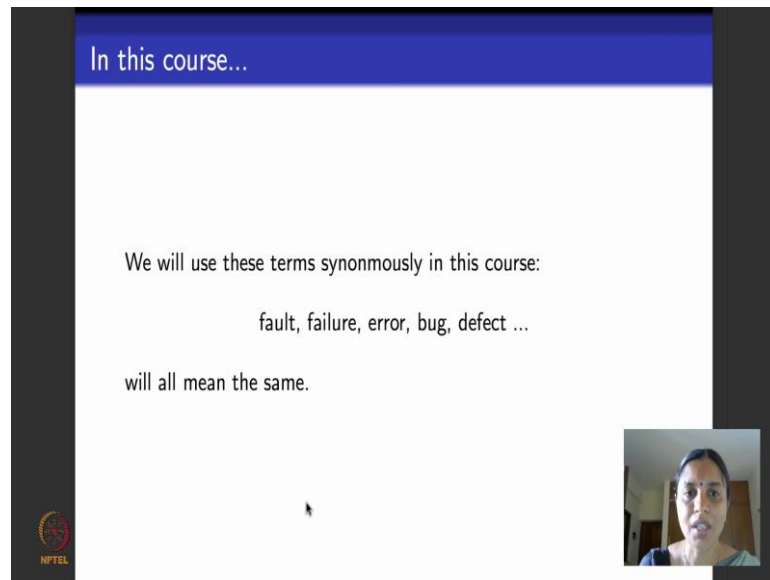


The slide is titled "Historical perspective" in a blue header. It contains two bullet points. The first bullet point discusses the terms "bug" and "fault" as used by Edison, quoting him: "It has been just so in all of my inventions. The first step is an intuition, and comes with a burst, then difficulties arise, this thing gives out and [it is] then that bugs, as such little faults and difficulties are called, show themselves and months of intense watching, study and labor are requisite." The second bullet point discusses the term "error" as used by Ada Lovelace, quoting her: "An analyzing process must equally have been performed in order to furnish the Analytical Engine with the necessary operative data; and that herein may also lie a possible source of error. Granted that the actual mechanism is unerring in its processes, the cards may give it wrong orders." An NPTEL logo is visible in the bottom left corner of the slide.

- The terms **bug** and **fault** was used by **Edison**:
"It has been just so in all of my inventions. The first step is an intuition, and comes with a burst, then difficulties arise, this thing gives out and [it is] then that **bugs**, as such little **faults** and difficulties are called, show themselves and months of intense watching, study and labor are requisite."
- The term **error** was used by **Ada Lovelace**:
"An analyzing process must equally have been performed in order to furnish the Analytical Engine with the necessary operative data; and that herein may also lie a possible source of **error**. Granted that the actual mechanism is unerring in its processes, the cards may give it wrong orders."

This is a small piece of history thanks to the book by Ammann and Offutt software testing. So, they connected the term bug and fault to Edison. So, apparently Edison use these terms first. So, he talks about his inventions and he says there is a lot of excitement when you initially invent something and then comes a bug or a fault, which he calls difficulties and he says they show themselves and it takes sometimes lot of effort to fix them. The term error is credited for its first use, to the first programmer who could have heard of namely the lady Ada Lovelace. So, when she was programming using Charles Babbage's analytical engine and punch cards, she says that, there could be an error. So, this is considered the first use of error and she says error could make the cards give wrong orders.

(Refer Slide Time: 07:00)



In this course...

We will use these terms synonymously in this course:

fault, failure, error, bug, defect ...

will all mean the same.

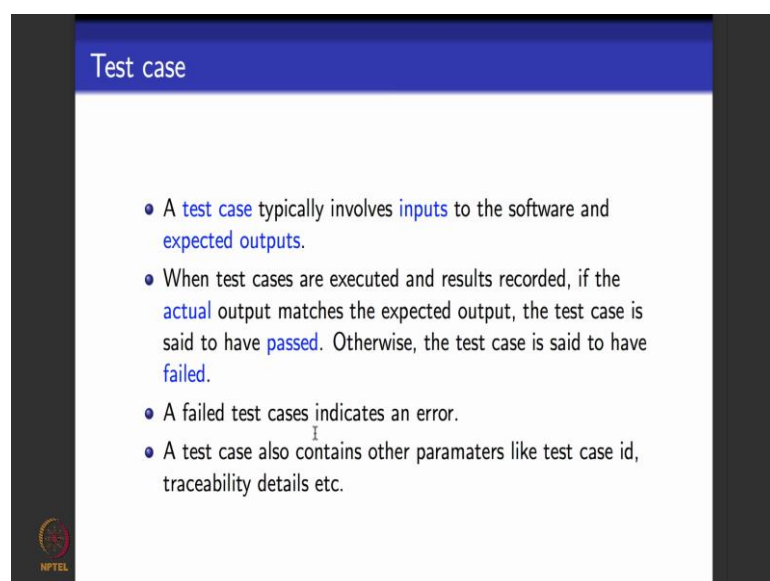
NPTEL

A small video inset in the bottom right corner shows a woman with dark hair, wearing a light blue top, speaking.

As far as this course is concerned, we will use all these terms synonymously--- fault, failure, error, bug, defect they will all be used interchangeably and synonymously, they will all mean that they something wrong in the software artifact that I am testing.

Another important term that you need to know and get a clarified right now that we will use throughout the course in testing is when we say testing the first thing that we have to do is to write test case. What is a test case?

(Refer Slide Time: 07:26)



Test case

- A test case typically involves inputs to the software and expected outputs.
- When test cases are executed and results recorded, if the actual output matches the expected output, the test case is said to have passed. Otherwise, the test case is said to have failed.
- A failed test cases indicates an error.
- A test case also contains other paramaters like test case id, traceability details etc.

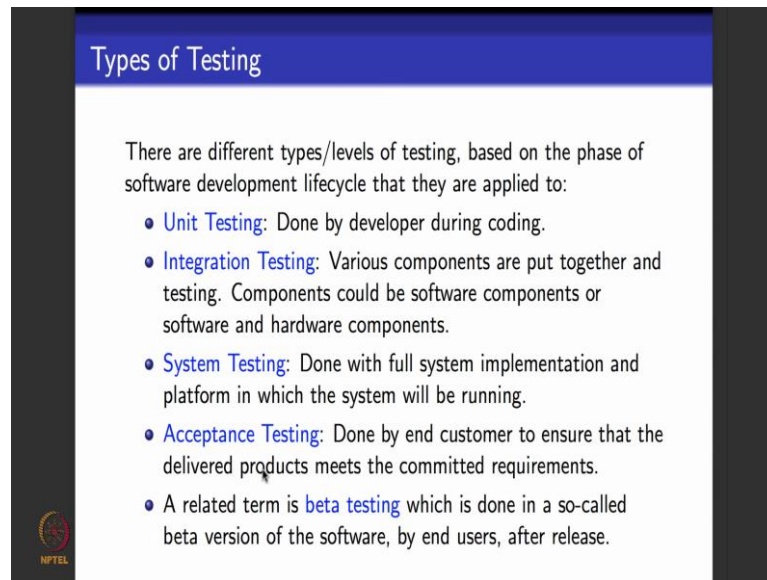
NPTEL

Test case typically has 2 main parts, it gives inputs to the software artifact that is concerned and it also says when I give this input to the software and execute the software on that, what is my expected output. So, test case has an input and what is the expected output, it is to be noted that if a test software artifact like code I always give only inputs to the codes and the part of the test case and be typically do not give values to internal variables to output variables and so on.

So, now I give this test case give inputs to the software and run the software, the software produces the output right. So, the output produced by software is what is called actual output. So, now, what we do typically, we check the actual output, compare the actual output to expected output and say whether they match or not. If the actual output matches the expected output, then you say the test cases said to have passed. If the actual output differs from the expected output then typically in testing this says that the test case has failed. A failed test case indicates that there is a fault in the software and the fault has manifested itself by resulting in an error, as per the terminologies that we saw.

Typically test case also contains things like an ID because you need to track and record the test case and people also maintain traceability information. What is traceability for a test case? Traceability tells you what are you testing this particular software artifact for. You might be testing a particular functionality in a piece of code. Who tells you to test for such functionality? Where did that functionality come from? That functionality could have come from a set of requirements. So, if it came from a set of requirements, which are the requirements that it come comes from? So, all these information is maintained in what is called traceability matrix, traceability data which is also a part of the test case.

(Refer Slide Time: 09:34)



Types of Testing

There are different types/levels of testing, based on the phase of software development lifecycle that they are applied to:

- **Unit Testing:** Done by developer during coding.
- **Integration Testing:** Various components are put together and testing. Components could be software components or software and hardware components.
- **System Testing:** Done with full system implementation and platform in which the system will be running.
- **Acceptance Testing:** Done by end customer to ensure that the delivered products meets the committed requirements.
- A related term is **beta testing** which is done in a so-called beta version of the software, by end users, after release.

NPTEL

Now, we will move on to looking at the various terms that are popular in testing and trying to gain clarity on what is what. So, when I do testing, I do testing throughout software development and at each stage of software development I could do a different kind of testing. So, there are various levels of testing based on which phase of software development that I am testing on. The right first testing that I will do is called unit testing, unit testing is typically done by a developer. When he or she writes the code, they debug the code, they test the code then and there to see if the code is working fine as for as its requirements of concern.

These days because of the stress on agile methodologies developers are expected to do a lot of unit testing themselves, they do not really have the luxury to pass things to a tester and say that you do it, right?

After unit testing we typically do what is called integration testing. So, what is integration testing? A developer again can do integration testing. So, let us see a developer has written several modules of code and then they are trying to put together the code, what do you mean by putting together the code? Maybe the code is in different methods in particular method could call another method, a procedure could call another procedure. So, when you test these focusing on these calls, these function calls, procedure calls when you test the interfaces that occur between the various modules in code then you do what is called software integration testing.

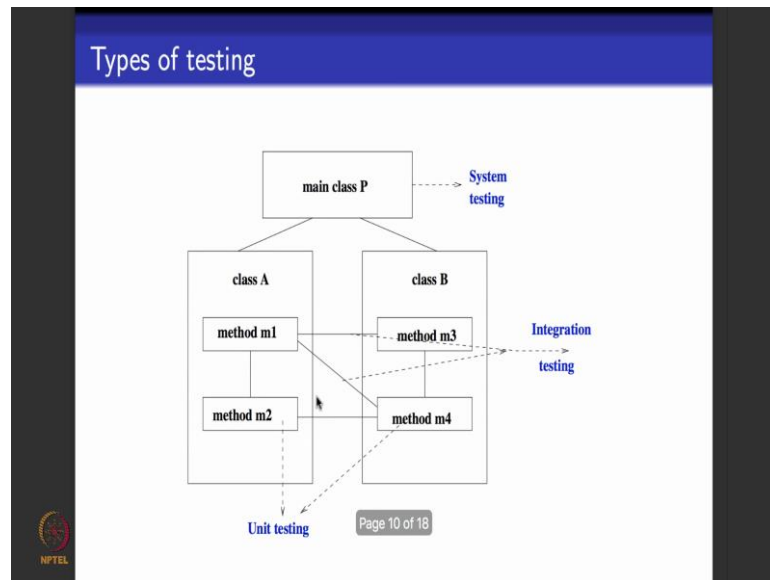
There is also another terminate integration testing, where people take the software as it is written and put it on the hardware that is supposed to work on. Let us say you have a fairly large server right you put it on the server that is supposed to work on you supposed to have integrated the hardware within the software. So, when you integrate the hardware with the software and test the integrated piece of hardware and software that is also called integration testing. So, post integration testing, the next step that people do is what is called system testing. So, in system testing the entire system is put together. Let us say take the case of an enterprise software, you will put together the server, the database, the web interface, the application server, the clients and all the interfaces between them and then you will test the end to end system from where the inputs come, what happens in the central system, what are the kind of main decisions, that are taken and how do they result in the output being produced.

In embedded software typically like a car or a plane you put the software as a part of the main system, which could be within the car or within the plane, the inputs will come from sensors the main control algorithms will run, from by picking up the sensor and puts from the bus and then they will produce actuator output. So, this end to end software as a part of the system with the inputs and outputs integrated from their original sources, the entire system when we tested we call it system testing. After system testing we believed that the system is more or less ready for reduce.

The last phase of testing the people do is what is called acceptance testing. Here you give the system to an end user and check if the system is working fine as per the end user committed requirements that the software or the system was supposed to meet. You might have also heard of a term called beta testing. What is beta testing? In beta testing people specifically release what is called beta version of software. When they release a beta version of software they roughly mean that this software is working fine, but I may not have tested it for mitigating all the involved risks.

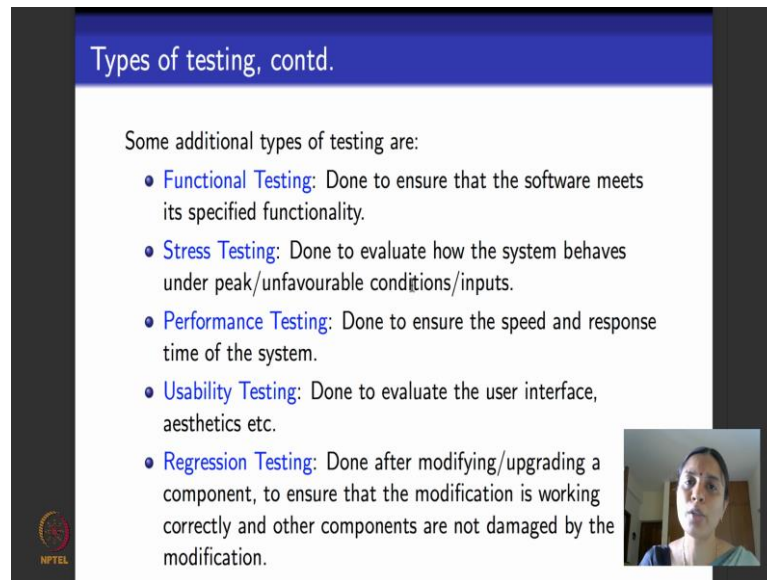
So, they are asking the users to start using the software and let them know if it has any bugs or defects. So, when you do this kind of testing then you are doing what is called beta testing right.

(Refer Slide Time: 13:27)



So, let us take an example to understand is a little better. So, let us say there is a main class called class P, which in turn has 2 classes A and B. Let us say class A has 2 methods m1 and m2, class B also has 2 methods m3 and m4. Let us see a particular developer is working on writing code for method m2 or m1 or m3--- just that method. So, when the developer finished his writing code or while writing code the kind of testing that the developer would do to test the functionality of just that method is what is called unit testing. So, now, if you look at this picture method m1 happens to call methods m3 and m4, m3 intern calls method m4. So, when a developer puts together these 2 classes all their methods and tests features specific to these calls, then the developer is doing what is called software integration testing.

(Refer Slide Time: 14:41)



The slide is titled "Types of testing, contd." and lists five additional types of testing:

- **Functional Testing:** Done to ensure that the software meets its specified functionality.
- **Stress Testing:** Done to evaluate how the system behaves under peak/unfavourable conditions/inputs.
- **Performance Testing:** Done to ensure the speed and response time of the system.
- **Usability Testing:** Done to evaluate the user interface, aesthetics etc.
- **Regression Testing:** Done after modifying/upgrading a component, to ensure that the modification is working correctly and other components are not damaged by the modification.

The slide also features the NPTEL logo in the bottom left corner and a small video inset of a woman in the bottom right corner.

Let us say this after this integration testing, the developer puts together the entire system and tests at the level of the main class B then that phase of testing is what is called system testing. You might have heard of all these additional terms in testing: functional testing, stress testing or load testing, performance testing, usability testing. What are they we will see them now. So, what is functional testing? Functional testing is typically done to ensure that the software meets its specified functionality. What do we mean by this? Let us take a software that runs as a part of an ATM machine of a bank right. What do you think is the core functionality of an ATM software? Core functionality of the ATM software could be the following. Once the user enters the password and the pin, if the credentials match and if there is sufficient balance left in the users account then, the amount that he or she has requested to be withdrawn should be provided to the user correctly.

So, ATM is meant to do just this and when I test a software to check if it needs its main intended functionality then I do what is called functional testing. Another kind of testing that is popularly done is what is called stress testing. So, here what happens is that the system is meant to be up and running in available lot of times and sometimes the system experiences what is called peak input conditions. For example, recently the class 12 marks were released right? So, they were released on the internet through a web application software. So, this system will experience what is called stress levels of input within 3-4 hours of the time in the date of release when all the students would want to

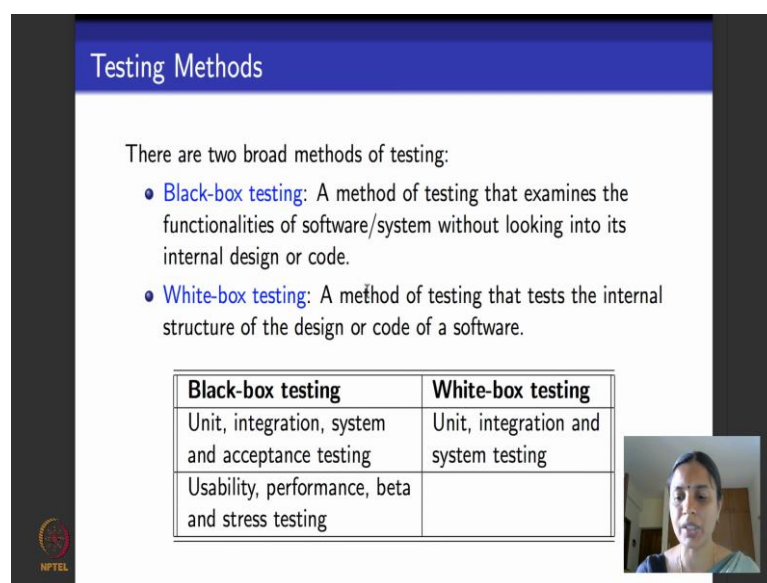
login and check what their marks are. So, when you test a system under this peak load conditions or input conditions then you do what is called stress testing.

The next kind of testing that you might have heard about is what is called performance testing. So, what will happens here is that here, people do testing to ensure that the system meets it is desired response time. It is fast enough, like for example, when we insert the card into an ATM machine, we want the ATM machine to be able to respond with the welcome screen almost immediately right, we cannot afford to wait for 10 seconds or minute or so. Performance testing specifies performance requirements on the system and checks whether all these performance requirements in terms of latency, speed, response, time, etcetera are met by the system.

Next popular kind of testing is what is called usability testing. It is not only useful to have a great software system, but the great software system is meant to be usable by a user which means the software should have a good user interface. It could be graphical or not, but the software should have a good user interface that is friendly. It should also be accessible, accessible in the sense by someone who is visually impaired, hearing impaired. It should be accessible. So, testing that is done to ensure that a software is usable, has a good interface, meets all the accessibility guidelines, is aesthetic enough---the kind of testing that is done to do all this is what is called usability testing.

Finally, one important term in testing before we move on is what is called regression testing. So, when does regression testing happen? So, let us see you have developed piece of software and usually released it. So, post release there is either change to the software that you do or you add a new functionality to the software. Now it is obvious that you might want it first check if the change or the new functionality that you have added is working fine and the second thing that you might want to do is that this change or the new functionality does not affect any of the other features that they are already present in the software before this was done. The kind of testing that is done to ensure that the software is working fine after modifying or upgrading it is what is called regression testing.

(Refer Slide Time: 18:45)



Testing Methods

There are two broad methods of testing:

- **Black-box testing:** A method of testing that examines the functionalities of software/system without looking into its internal design or code.
- **White-box testing:** A method of testing that tests the internal structure of the design or code of a software.

Black-box testing	White-box testing
Unit, integration, system and acceptance testing	Unit, integration and system testing
Usability, performance, beta and stress testing	

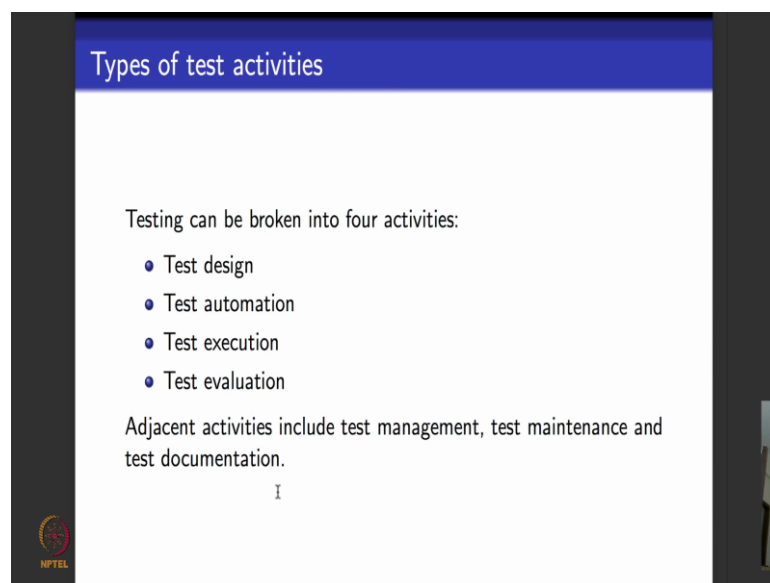
So, when all these methods of testing, all the various types of testing basically, 2 different methods in testing, you might have heard of these terms Black box testing and White box testing. So, what happens in black box testing? When you do black box testing you consider the software artifact that is being tested as a black box, which means you do not look inside it like, when you test the code you consider the entire code as a black box give inputs to the code execute the code and observe the outputs to check if the expected outputs match the actual outputs.

White box testing is the exact opposite. When I am testing a piece of code, with reference to white box testing I look into the code and I test for requirements like, suppose there is an if statement in the code can you write a test case that will cover the if statement. What is it mean to cover in the if statement? It means that can you write a test case which will execute the then part of the if statement ones which will execute the else part of the if statement once. So, similarly, when a white box test code I could insist that you cover loops in the code. So, let us say there is a while loop in the code, covering the loop would mean that you write test cases to skip the loop, to execute the loop in normal operations and execute the loop on boundary conditions.

So, in summary what does white box testing do? White box testing actually looks at the code, looks at the design at the corresponding software artifacts, looks at structure what it what is it have, what are the kind of statement, what are the kind of design elements it

has and then it tells you how to test it by looking at its structure. So, black box testing applies to almost all phases of testing, all types of testing it applies throughout the development life cycle, it applies for doing usability testing, performance testing, stress testing and so on. White box testing is typically done only during software development. Once software is ready put together the system is tested. Later on when we test for other non functional requirements like performance, stress, response time, good user interface, being accessible, we typically do not test it, as a white box testing then we consider the system as a black box and then test for all these quality features in the software right.

(Refer Slide Time: 21:12)

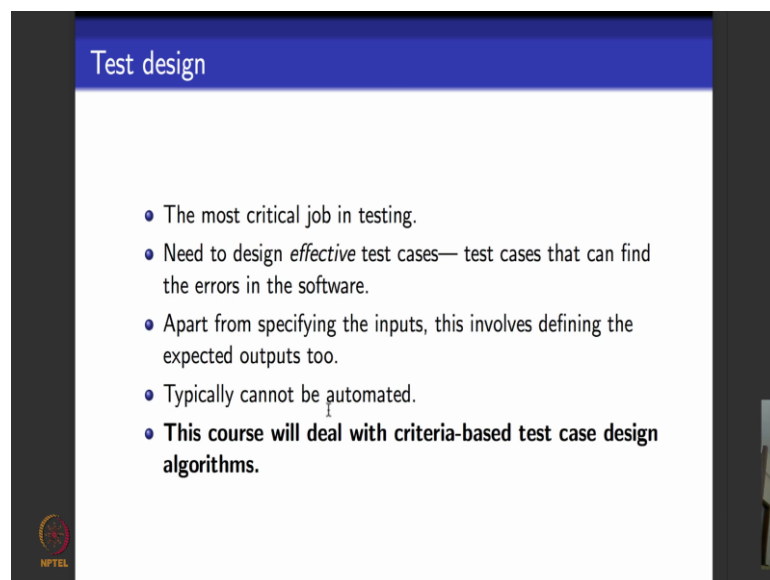


Now, we will move on to the last module, where we look at the various activities in testing. There are typically 4 raw activities in testing. So, we begin with design in cases because, if you do not have a test case then you cannot execute it and you cannot find errors in the software. The first thing is to be able to define or design a test case, once we design a test case I have to make it have to make it execution ready, that process is what is called test automation. After I make it execution ready, I can use a tool to actually executed and record the results. After I record the results then, I do what is called evaluation or analysis to check whether the test case is passed or failed and if it is failed and if there is an error where the error is and so on. These are the core technical activities and testing.

Apart from this there are several other umbrella activities. Like you have project management in software development, project management specific to testing is what is called test management. So, there are test managers to design test teams try to organize them.

Another important activity that happens is a part of testing is what is called test maintenance and test documentation. A lot of testing especially in enterprise software domain relies on reusing test , extensive number of test cases are reused again and again to be able to text software. So, how do I reuse a test case I should be able to document it and store it well to be able to make it amenable for use.

(Refer Slide Time: 22:59)



The slide is titled "Test design" in a blue header. It contains a bulleted list of five points. The first four points are in blue, and the fifth point is in black. The NPTEL logo is in the bottom left corner.

- The most critical job in testing.
- Need to design *effective* test cases— test cases that can find the errors in the software.
- Apart from specifying the inputs, this involves defining the expected outputs too.
- Typically cannot be automated.
- **This course will deal with criteria-based test case design algorithms.**

So, the kind of activities people do to ensure the test cases are well documented and well maintained and available for reuse as an when they are needed broadly constitute test maintenance and documentation.

So, now moving on to test case design this is considered the most critical job in testing. Why is it considered the most critical job in testing? Because, the Pareto principle applies to testing which means roughly 80 percent of the errors, a lot of the errors is focused on a very small percentage of the code. So, if I do not design my test cases effective what do you mean by effective in turn, if my test cases are effective then they will find errors faster, they will find all the errors or most of the errors areas that are

present in software, there is no point in saying that I designed so many test cases and I tested them for days and days and your software is doing fine right.

So, the effectiveness of test case design is in finding errors right and this needs domain knowledge. This needs knowledge about the system, about how it is developed and typically cannot be alternative. A lot of this course will deal with algorithms and testing techniques, that we will use to design test cases. I will give you more details about the precise kind of algorithms and artifacts that we will use in the next module.

After you designed your test case the next comes the converting this test cases in to executable script. So, you might have a test execution framework you could use JUnit, you could use Selenium. These are open source tools that will let you do test execution. So, you have to be able to make your test case ready for execution please remember, I told you that test case always talks about Inputs. So, suppose there is a requirement white box testing requirement, which says that you go to a while loop which is somewhere deep inside my code and you test for coverage of that while loop. It might so happen that the while loop has got no input variables, it directly deals with internal variables. Now it is up to the tester to be able to design test cases and give test cases input values such that this while statement is reached; not only it is this while statement reached this while statement is also covered under various coverage requirements.

So, how does a tester go about doing this? Software has to meet 2 important criteria called observability and controllability. We will look at these 2 concepts in the next module in this week. So, after your design your test case and it is ready for execution then comes the job of actually executing it and recording the results, this typically is almost always fully automated. There are several open source and proprietary tools that can do this really well for you.

After you have executed the test case, again, you need human intervention to be able to evaluate the result of the test case. So, let us say the test case is passed everything is fine, but if the test cases failed that indicates that the software is in an error condition. So, where exactly is the error? which is the erroneous components? Especially when you are doing system testing or integration testing, your software or system might be fairly large. It takes some amount of domain knowledge experience and human effort to be able to

isolate the fault and facilitate the development team to be able to debug and test it once again.

So, we would deal mainly with test case design algorithms in this course. So, in the next module I will tell you what are the algorithms are and what are the mathematical models modelling software artifacts that we would deal with in this course.

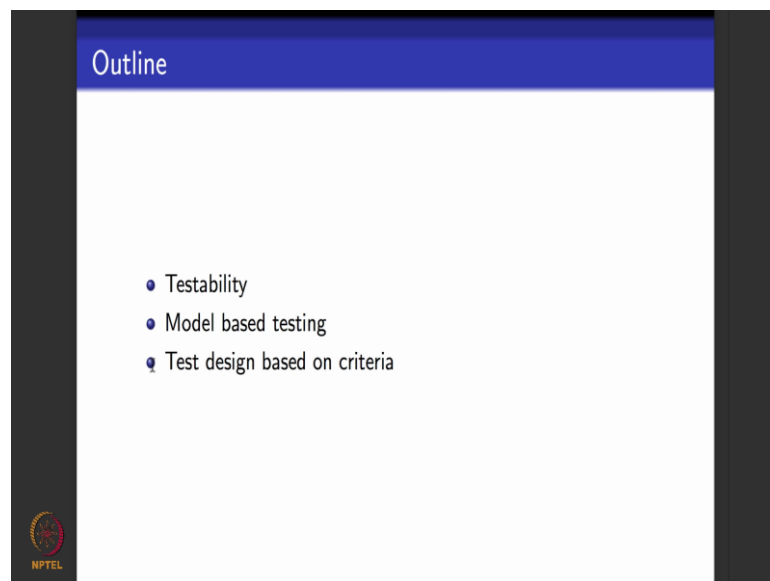
Thank you.

Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 03
Software Testing: Testing based on models and criteria

Hello everyone. This is the lecture on the second module of first week. So, what will be seeing in today's lecture? We will look at it what it means to test the software.

(Refer Slide Time: 00:22)

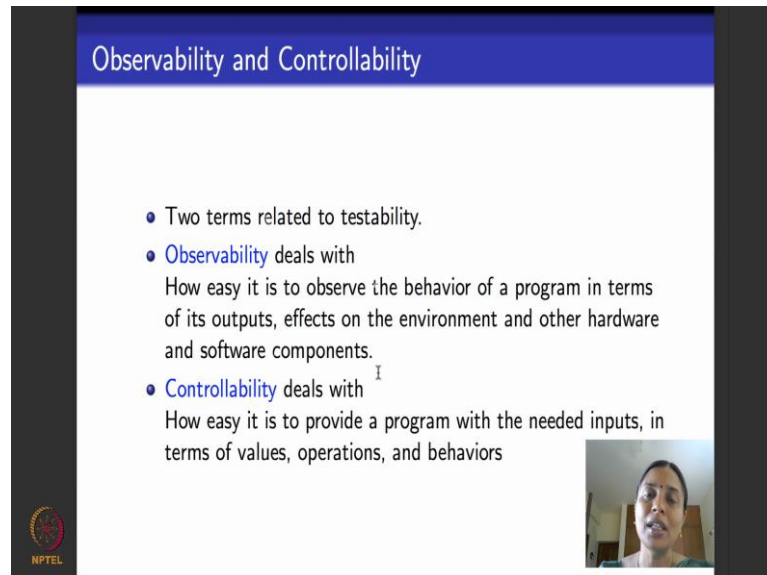


What is testability; we will also look at what is model based testing as its commonly perceive in the testing world, and how we will use model based testing when we design algorithms for test cases. As I told you lot of this course is going to be on algorithms and methodologies for test case design. And in fact we will be looking at what is call criteria based test case design. So, we will formally define what is criteria, and look at the various artifacts on which we will be defining these criteria to design test cases for.

So, I begin with introducing what is testability. So, what is testability? If you look at it in simple English terms it simply answers the question--- is the software testable or not. So now, if I ask the same question again in term what is it mean for the software to be testable, right? So, we say can I give inputs to the software; inputs a test cases to software? After giving the test cases to the software can I execute the software an observe the outputs. You might think it is a very obvious question--- when I write a piece

of core what is the difficulty about giving inputs to the software, executing the software and observing its outputs.

(Refer Slide Time: 01:37)



The slide is titled "Observability and Controllability" in a blue header. It contains a bulleted list of two terms related to testability. The first bullet point is "Observability" which deals with how easy it is to observe the behavior of a program in terms of its outputs, effects on the environment and other hardware and software components. The second bullet point is "Controllability" which deals with how easy it is to provide a program with the needed inputs, in terms of values, operations, and behaviors. In the bottom right corner of the slide, there is a small video inset showing a person speaking.

- Two terms related to testability.
- **Observability** deals with
How easy it is to observe the behavior of a program in terms of its outputs, effects on the environment and other hardware and software components.
- **Controllability** deals with
How easy it is to provide a program with the needed inputs, in terms of values, operations, and behaviors

So, we will see what are the particular hiccups that can arise while doing this. Two related notions of testability are what are called observability and controllability. So, what is observability mean? Observability it tries to answer the following question. So, it says the suppose you give a software certain kinds of inputs and execute it, how observable is its behaviour,? How observable are the outputs that the software produces?

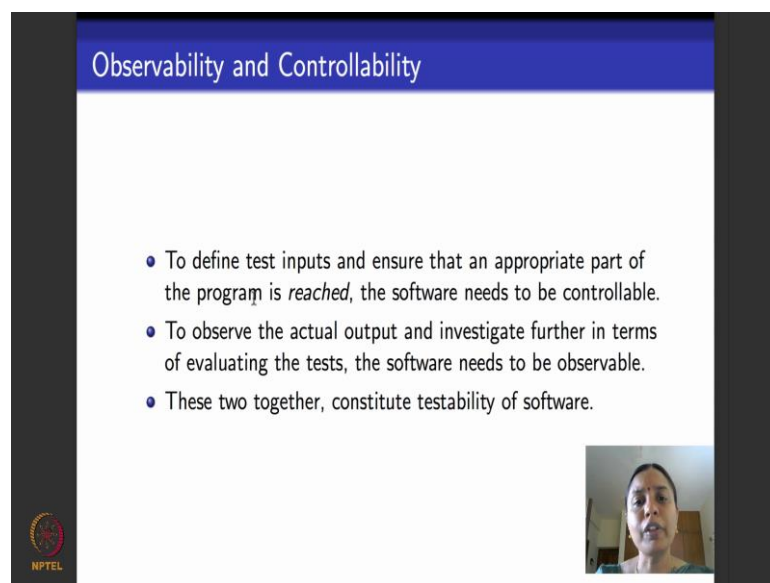
It could do we doing a lot of things silently and producing only the final output, whereas as a tester you might be interested in some intermediate outputs that the software produces. So, observability tries to answer this question about how observable the outputs produced by the software are and how much we can tweak the outputs that we want the software to produce towards testing it.

So, when I test the software I might want to observe a little more about its outputs, not actually the actual outputs of software. But I might want to know the values of specific internal variables, specific call and return values. Observability deals with all this. Another related term for testability is what is called controllability. What is controllability? Controllability says if a software controllable by giving it in some input and actually executing the software.

Again this might seem like a trivial question you might ask why is it so difficult to give inputs to a software. Let us take a particular case for example, where piece of software runs for some time and then at particular point and code it calls a particular function. So, I want to be able to let us say “integrate test” this function call. So, what I am interested in controllability is the fact that I should be able to give inputs to the softwares such that I can ensure that the particular function of procedure that is embedded deep inside the code is actually called for the values of the inputs. And not only is the function called, I now want to be able to observe what the function returns in turn.

So, controllability and observability help you to answer these questions and software. And, they are one of the several different quality attributes broadly call “ilities”, that the people worry about while testing and writing software.

(Refer Slide Time: 03:51)



The slide is titled "Observability and Controllability" in a blue header. It contains a bulleted list with three points. In the bottom right corner, there is a small video inset showing a woman speaking. The NPTEL logo is in the bottom left corner of the slide area.

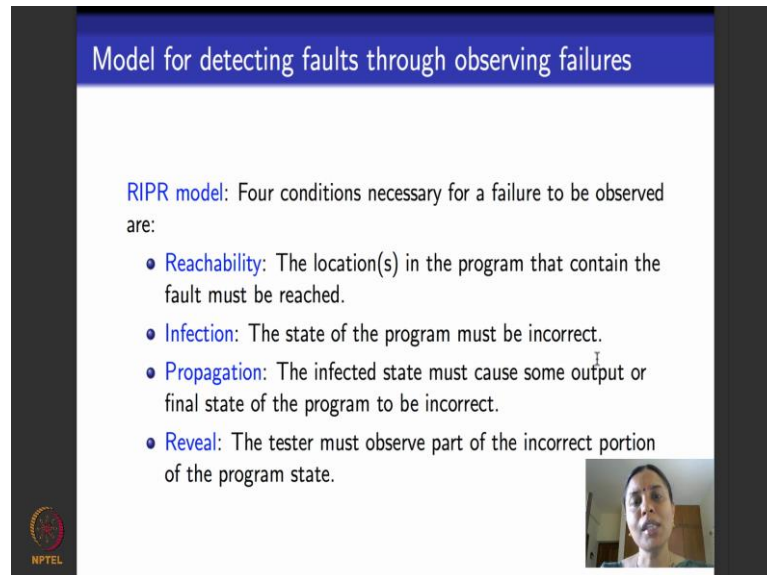
- To define test inputs and ensure that an appropriate part of the program is *reached*, the software needs to be controllable.
- To observe the actual output and investigate further in terms of evaluating the tests, the software needs to be observable.
- These two together, constitute testability of software.

So, to define test inputs and to ensure that in appropriate part of the code is reached, like I told you an appropriate function is called and I will be able to in turn observe the value return by the function I say that the software needs to be controllable. When I want to observe the value that is return by the function and see how it gets passed on to the main callee program then I say that the software needs to be observable. Observability and controllability put together constitute testability of a software.

Obviously, for a software to be testable, its testability quotient must be very high; it should be observable, it should be controllable. So, we will little more in detail

understand what this notion are; when people say a software is testable they usually talk about four parameters for observability and controllability put together.

(Refer Slide Time: 04:38)



Model for detecting faults through observing failures

RIPR model: Four conditions necessary for a failure to be observed are:

- **Reachability:** The location(s) in the program that contain the fault must be reached.
- **Infection:** The state of the program must be incorrect.
- **Propagation:** The infected state must cause some output or final state of the program to be incorrect.
- **Reveal:** The tester must observe part of the incorrect portion of the program state.

NPTEL

So, these four parameters are what are called RIPR model. So, what is R? R stands for reachability. Reachability mean suppose I am testing, white box testing, a piece of software and I want to check whether a particular statement is reached, whether a particular function call is reached, whether a particular code segment is reached; I should be able to give test inputs that guarantee the execution of software in such a way that this particular target place in the software of goal is reached.

Now the next important thing is I: I stands for infection, what do we mean by infection? Not only do I want to reach that place I want to be able to execute those statements in the software. And suppose there is an error that is observed I want the error to be exercised or I want the error to be reached; I want the software to reach its faulty state. And I want to be able to give inputs in such a way that I can infect that particular statement in the software and actually pull out the error or the faulty states the software was in. Suppose I manage to do that, but the software still manages to run fine and execute such that it acts normally, then even though the software is erroneous I, as a tester have not really observed the error.

So, the next parameter that I am looking for is P which stands for propagation. So, I say not only much particular statement be reached and the error be revealed as infected; the

error should also be propagated to output, observable output, such that I know that there is an error that has occurred in the software. And it could be propagated and for some simple reason like you might just forgotten to write a print statement or forgotten to track this particular values the propagated error might not actually get revealed to the tester. So, at the last parameter in this RIPR model that I am looking for is for the error to be revealed; this tester will be able to observe the incorrect part in the program and correctly isolate and identify which part of the code was the error in.

So, to summarise we say that if I am targeting a particular a piece of code that I suspect to be erroneous or that I want coverage to be achieved; somebody told me you please test this piece the code thoroughly. You please test this function and the function called thoroughly. So, I am looking for four properties the software should satisfy for it to be observable and controllable.

The first one is that the particular target code fragment or segment should be reachable. The particular target code fragment or segment, if it has an error should infect and the particular test case that I give should make sure that the error state has actually occurred. This occurred error, in turn, should propagated to the software producing a different kind of output that reveals the error. And the revealed error should, in turn, help the tester to be able to identify and isolate the error as having occurred in that piece of software. So, software that needs this RIPR model property is supposed to be highly testable.



(Refer Slide Time: 07:48)

RIPR example

Consider the code segment below:

```
input x,y;  
if (x < 10)  
{ z = x+1;  
  if (y < z)  
    --- error ---;  
}
```

- Reachability: True, any value of x can reach the first if statement. $x < 10$ will reach the second if statement.
- Infection: $x \neq 10$ will test the first if statement. $y < x+1$ will test the second if statement.
- Propagation: $x < 10$ and $y < z$ will result in reaching the error statement.



So, here is an example of an RIPR model: you consider the small code fragment that is given here I am not given the full piece of code here, I have just given a small fragment of the code. In this small fragment of the code there are two inputs x and y , and then they could be other statements lot of other commands and the software. But let us focus on this particular thing fragment of the code which says that for there are two nested if statements here. The first statement checks if x is less than 10 that it sets the value of z to be x plus 1. And then there is one more if inside that which says that if y is less than z then there is an error in the software. I haven't actually written what the error could be, I have just written it like a stub which marked it out as error.

So, suppose I want to actually; my goal is to be able to be actually reach this error statement and reveal the fact that this error as occurred. So, what does RIPR parameters do for this? So, let us look at reachability first. So, what is it mean for the first if statement to be reachable; any value of x will reach that statement assuming that there are no code fragments that change the value of x , any value of x that is given here will reach the first statement which is x less than 10. So, reachability can be thought of as being true, always true, there is nothing specific that I need to do.

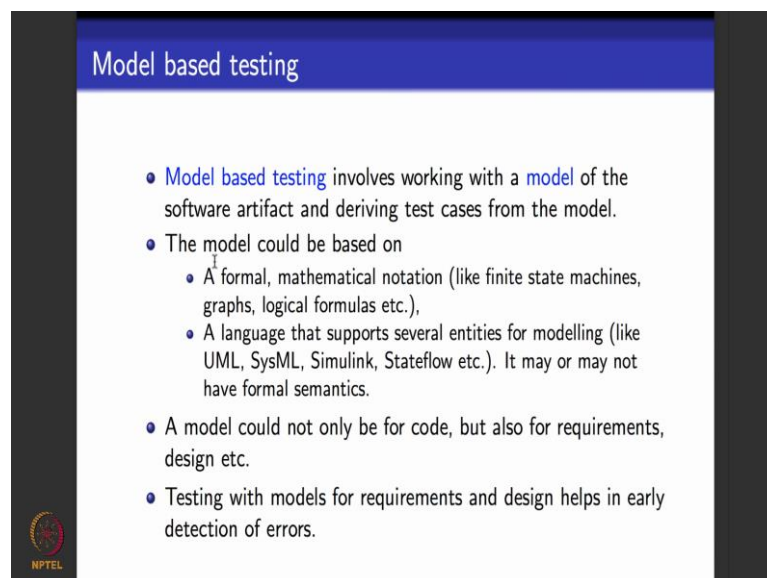
But to be able to reach the second nested if statement; suppose the first if statement test negative in the sense suppose I give a value of x that is not less than 10. So, this if statement, this predicate will written false and the code will exist this if statement it will never enter this if statement. If it does not enter this if statement it is not going to be able to test the second if make it positive and then reach the error statement. So, reachability for the second if statement will happen only if the first if statement returns true and the first if statement will return true if I give a value of x that is less than 10.

So, the predicate x less than 10 should be true for me to be able to reach the second if statement. So, what is infection? Infection means I not only reach the statement, I manage to execute that statement and make it true or false. So, again an infection for the first if statement is any value of x less than 10 like for example, or any value of x greater than 10 or even x is equal to 10 will manage to infect this first statement. For the second statement remember z is set to x plus 1. So, you must give a value of y that is less then x plus 1 to be able to test the second if statement. Now moving on, when it comes to propagation I should be able to go past the first if statement makes it true, go past the second if statement make that also true, and then only I will reach the error statement.

So, both these predicates x less than 10 and y less than z will have to be true for me to be able to reach the error statement.

I hope what reachability infection and propagation is very clear. So, in this piece of the example it is quite small, so it easy to be able to do what are the conditions for reachability, infection and propagation and so on. But for large fragments of code, where I have thousands of lines of code it is not an easy problem to be able to come up with less list of predicate for reachability for infection and propagation and to be able to solve them. In later modules when we look at testing criteria we will see how what it means to solve the predicate or a particular piece of code for reachability, for infection, and for propagation in detail.

(Refer Slide Time: 11:20)



The slide is titled "Model based testing" in a blue header. It contains a bulleted list of points. The first point states that model based testing involves working with a model of the software artifact and deriving test cases from the model. The second point states that the model could be based on two sub-points: a formal, mathematical notation (like finite state machines, graphs, logical formulas etc.), and a language that supports several entities for modelling (like UML, SysML, Simulink, Stateflow etc.). It may or may not have formal semantics. The third point states that a model could not only be for code, but also for requirements, design etc. The fourth point states that testing with models for requirements and design helps in early detection of errors. In the bottom left corner, there is a small NPTEL logo.

- Model based testing involves working with a model of the software artifact and deriving test cases from the model.
- The model could be based on
 - A formal, mathematical notation (like finite state machines, graphs, logical formulas etc.),
 - A language that supports several entities for modelling (like UML, SysML, Simulink, Stateflow etc.). It may or may not have formal semantics.
- A model could not only be for code, but also for requirements, design etc.
- Testing with models for requirements and design helps in early detection of errors.

I will now move on to looking at model based testing. What is model based testing mean? In the English sense of the term, model based testing means that you have a model of the software artefact; the artifact could be requirements it could be design, it could be architecture, it could be code, and then you design test cases by looking at the model. There are of course, several other uses of the models for software, well done models could be used to auto generate codes they could be used to early validate the system. There are several other uses for model based design and model based testing.

There are broadly two kinds of models: the model could be based on a formal, mathematical notation. Here are some popular modelling languages: finite state

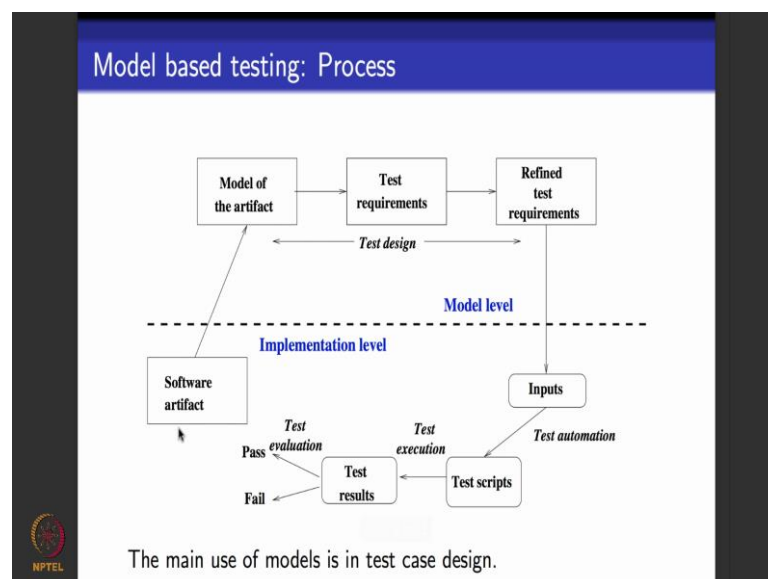
machines, graphs, logical formulae. As I told you in the last module there are several different verification techniques, like model checking theorem proving, they all use mathematical models of software artifacts to be able to verify a software.

Models it could also mean a particular domain language or specifically designed language that specifies all the artifacts that particular software is concerned. You might have heard popular modelling language like UML which stands for unified modelling language, SysML which is systems modelling language, Simulink, Stateflow which are modelling languages that are proprietary to a company called Mathworks which makes a popular tool called MATLAB.

So, these modelling languages support several diagrammatic notations for me to be able to module various software artifacts like use cases requirements design, control laws and so on. They may or may not have formal semantics. So, it is to be noted as I told you that the model need not be only for code; it could be for requirements, it could be for design, and so on. Testing with models for requirements and design: suppose I manage to have good, well defined models for requirements and design and I am able to design test cases for them, then remember that I am doing this kind of testing right at the requirements level or design level, before I write code.

So, if I find an error I find it early on in the life cycle and that is considered to have lots of benefits. So, what is the process of model based testing, how does it work?

(Refer Slide Time: 13:32)



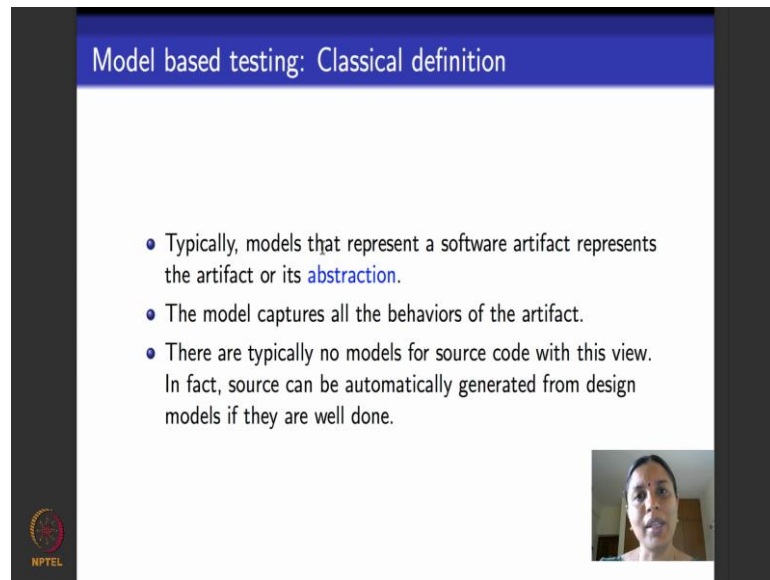
Let us begin here with this part: I have a software artifact with me under consideration that would be code that could be a requirements document that could be a design document. Code is typically executable, but many software companies have requirements of design document written in English. So, English documents cannot act as models. So, I take the software artifact and come up with the model of the artifact. The model of an artifacts is designed in a language that I have predisposal decided as suitable for the class of software that I am working with.

Once I have the model of this software artifacts I do what is called model based design and I can subject this model to several different uses, but what we will focus on today, is in this particular slide, is the use of the model when it comes to designing test cases. So, I take this model of the artifacts and then I also have a set of test requirements that have been given to me. Test requirements could say something like--- there are some highly critical requirements you please design test cases to exhaustively test for all the requirements. Test requirement could say something, like you cover this particular fragment of the model or a piece of code.

So, I take these test requirements and refine them with reference to the model that I have in mind. After I refine the test requirements I have done my test case design and I am ready to give my test case as inputs. Remember once I have a test case that has to be actually executed in code. So, I assume that I have an implementation ready and I move on to the implementation of the software. Once I have given my test case design I pass these inputs, make it ready for execution; make it ready for execution you remember in the last module we saw this step called test automation. So, I do test automation and get that test scripts which are ready for execution then I use the tool to execute this test scripts and observe the results. And then I conclude the test cases have passed or failed.

So, this is the normal process of testing that we saw the only difference here is that while doing designing of test cases I have a particular model of the software and hand and I design test cases for a set of test requirements on that specific model of the software. Here my main use of models is to be able to do test case design.

(Refer Slide Time: 16:03)



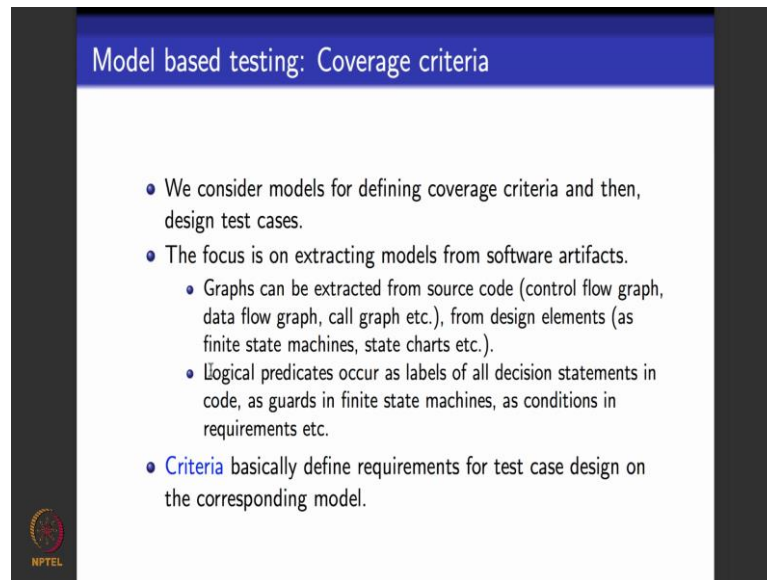
The slide has a blue header with the text "Model based testing: Classical definition". Below the header, there is a bulleted list of three points. In the bottom right corner of the slide, there is a small video inset showing a person speaking. In the bottom left corner, there is a small logo for NPTEL.

- Typically, models that represent a software artifact represents the artifact or its **abstraction**.
- The model captures all the behaviors of the artifact.
- There are typically no models for source code with this view. In fact, source can be automatically generated from design models if they are well done.

So, the classical view of model based testing or model based design is to say that models could represent a very high level abstraction of the software, it need not represent the actual code. In fact, code is almost always never represented as a model. Code is represented in the programming language of your choice and is always an executable entity. Models may or may not be executed, they could be static artifacts that represent the code. Typically a model is supposed to capture all the behaviours of the particular software artifacts and it is for the user to ensure that the model of the software is accurate and good enough to meet the required expectations of the model. If the model that you do of the software artifacts is itself wrong, then obviously the test case and everything else that you do with the model is also going to be wrong.

So, is up to the user to ensure that he or she knows the modelling notation well and is modelled software well enough to be able to design all the test cases. So, we do not view the classical view of model checking in this particular course, we will view model based testing for doing coverage criteria based design in this course.

(Refer Slide Time: 17:13)



The slide is titled "Model based testing: Coverage criteria" in a blue header. It contains a bulleted list of points. The first point states that models are considered for defining coverage criteria and then design test cases. The second point states that the focus is on extracting models from software artifacts, with sub-points mentioning extraction from source code (control flow graph, data flow graph, call graph etc.), from design elements (finite state machines, state charts etc.), and logical predicates occurring as labels of all decision statements in code, as guards in finite state machines, as conditions in requirements etc. The third point states that criteria basically define requirements for test case design on the corresponding model. The NPTEL logo is visible in the bottom left corner.

- We consider models for defining coverage criteria and then, design test cases.
- The focus is on extracting models from software artifacts.
 - Graphs can be extracted from source code (control flow graph, data flow graph, call graph etc.), from design elements (as finite state machines, state charts etc.).
 - Logical predicates occur as labels of all decision statements in code, as guards in finite state machines, as conditions in requirements etc.
- **Criteria** basically define requirements for test case design on the corresponding model.

So, what do we do? We first focus and represent the software artifacts as models, we typically work with several different models. So, where do these models come from? So, graphs are one kind of models that I want to work with. Where do I get graphs from? I can get graphs from source code. You might have heard of control flow graph or a flowchart corresponding to a piece of code. In fact, there is something called data flow graph which represents the control flow along with tracking of the variables that are involved at a particular piece of statement in the code.

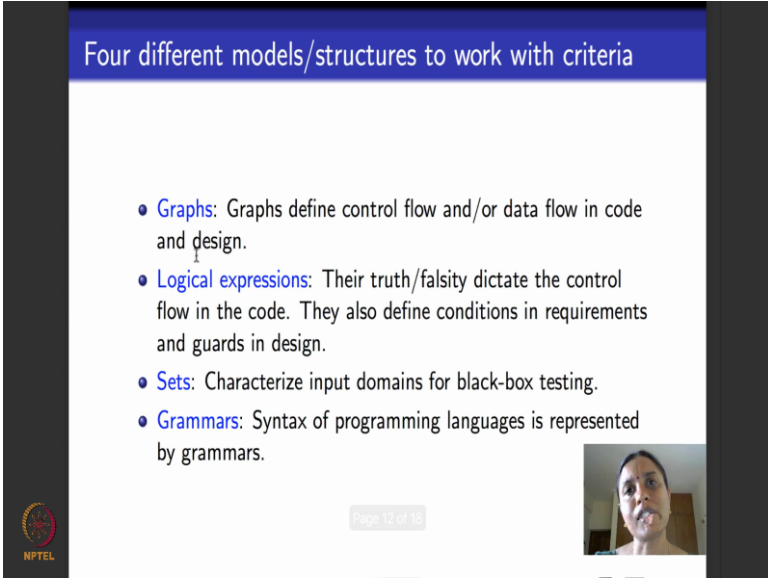
And then there is an notion of call graph in an inter procedural call graph when the code is modularized into several functions or procedures or method. I can also get graph from design; lots of designs these days are represented using UML notations which are basically finite state machines, state charts, all these are some specific kinds of graph. They have several different specific parameters to each of them, but they are all basically graphs.

Another class of models that we will be working with what are called logical predicates. Where do logical predicates come? If you see a typical code or an implementation is studded with them, at every decision point in the code. What is the decision point in the code? It could be an if statement, a while statement, a for statement which says that there is going to be some kind of the branching in the execution of a code, some kind of the choice in the execution of code. At every decision point in the code there is a predicate.

Based on whether the predicate is true or false the code execution takes one path or the other.

So, what we will try to do is we will try to work with graphs, logical predicates and a few other things is a models of software and design test cases based on these models, based on certain criteria.

(Refer Slide Time: 19:09)



The slide is titled "Four different models/structures to work with criteria". It contains a bulleted list of four items:

- **Graphs:** Graphs define control flow and/or data flow in code and design.
- **Logical expressions:** Their truth/falsity dictate the control flow in the code. They also define conditions in requirements and guards in design.
- **Sets:** Characterize input domains for black-box testing.
- **Grammars:** Syntax of programming languages is represented by grammars.

The slide also features the NPTEL logo in the bottom left corner, a small video feed of a speaker in the bottom right corner, and a page number "Page 12 of 18" in the center.

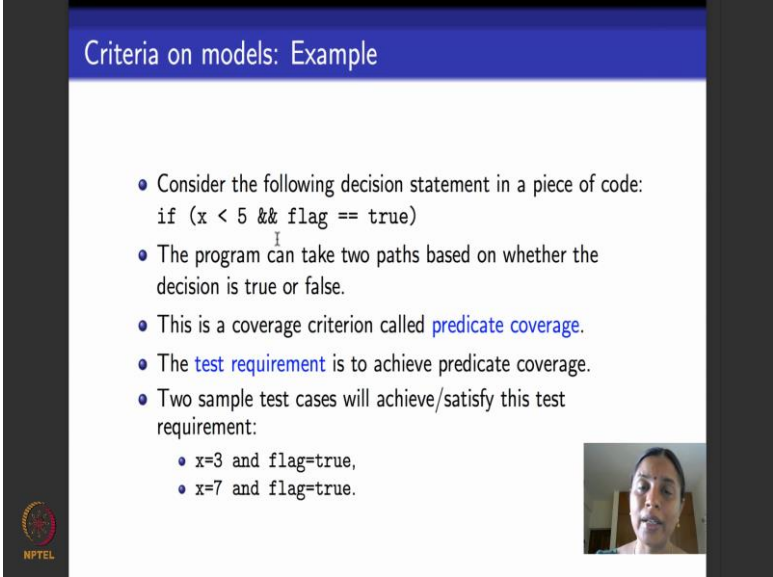
So, what are the four different model structures is that we will be working with as a part of this course. We will first begin with graphs next week. So, graphs typically in software define control flow, they also define data flow and as I told you little earlier they also could be graphs that depict calls of procedures, one procedure calling the other. So, we look at graphs as models representing such entities about a piece of software and then we will define criteria based on that and understand how to design test cases based on criteria.

Then we will look at logical expressions as they occur software, as they occur code, as they occur guards in design that represented as finite state machines, as they occur in requirements. And we will design test cases that check when a logical predicate could be true or when a logical predicate could be false. We will also, for the point of view of black box testing, look at sets of inputs that are given to a software and design test cases based on partitioning of those sets. So, sets would be another structures that we would be working with in this course.

And finally, when we look at coverage criteria we will also look at the underline syntax of programming language. So, every programming language as you know it could be C, it could be Java, it could be Python as an underlining grammar which tells you what is allowed in terms of writing in that programming language, how to write programs in the programming language. And when I compile a piece of software in the process of compiling I also check at the particular program adheres to the underline syntax or not. So, you can do mutation testing which exploits the syntax of a particular piece of programming language and designs test cases by manipulating inputs that adhere to the underline syntax or grammar of that particular language.

So, to summarise this slide, what we will do is we will look at software artifacts as four different structures or models: we will first consider them as graphs, and then will consider the logical predicates that occur in the code, then we will look at sets of inputs and outputs, and finally we will look at the grammar or the syntax of the programming languages. For each of these, we will design what are called testing criteria and also see algorithms on how to design test cases to test for these criteria.

(Refer Slide Time: 21:39)



The slide is titled "Criteria on models: Example" in a blue header. It contains a list of bullet points explaining predicate coverage. The first bullet point shows a code snippet: `if (x < 5 && flag == true)`. The second bullet point states that the program can take two paths based on whether the decision is true or false. The third bullet point identifies this as a coverage criterion called "predicate coverage". The fourth bullet point states that the "test requirement" is to achieve predicate coverage. The fifth bullet point lists two sample test cases that will achieve/satisfy this test requirement: `x=3 and flag=true,` and `x=7 and flag=true.` In the bottom right corner of the slide, there is a small video inset showing a woman speaking. The NPTEL logo is visible in the bottom left corner of the slide.

- Consider the following decision statement in a piece of code:
`if (x < 5 && flag == true)`
- The program can take two paths based on whether the decision is true or false.
- This is a coverage criterion called **predicate coverage**.
- The **test requirement** is to achieve predicate coverage.
- Two sample test cases will achieve/satisfy this test requirement:
 - `x=3 and flag=true,`
 - `x=7 and flag=true.`

So, here is a simple example to understand what criteria means. Let us say somewhere in your piece of code there is this following decision statement. How does this read? It says if x is less than 5 and a Boolean variable called flag is true then you do something. We are not interested in what we do, I just want to focus on this particular if statement.

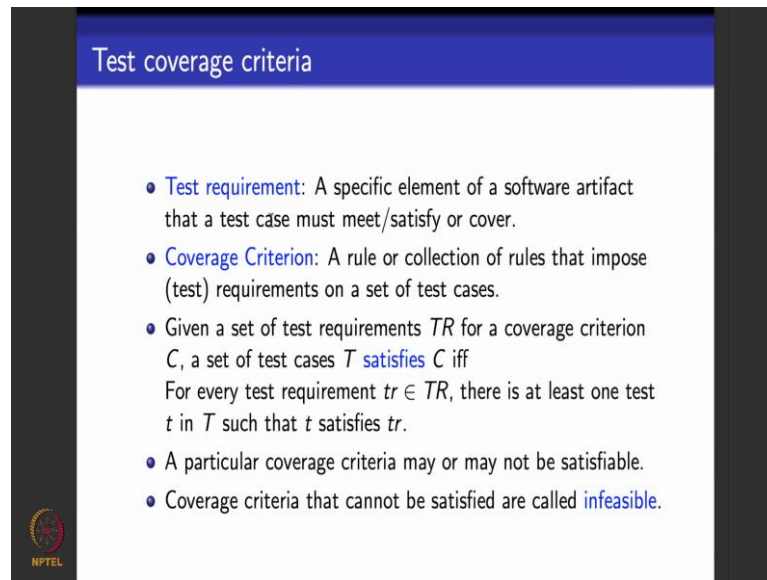
Now, what is the label of this if statement? The label of the statement is this particular thing right $x < 5$ and flag is equal to true. It involves two kinds of variables: it involves, maybe, an integer variable x whose values compared to 5, and it involves a Boolean variable flag which is tested to be equivalent to true. So, this whole thing is what we call a predicate. So, let say suppose this predicate turns out to be true that is x is indeed less than 5 and flag is also equal to true then you say that the if statement takes the then branch of the code. And suppose this predicate is false then you say that the if statement takes what is called the else branch of the code.

So, how will I make this predicate true? I say, you write two kinds of test cases one that makes this predicate true and one that makes this predicate false; that way I would have exercised this if statement for predicate coverage or branch coverage. When I cover a predicate I also cover then branch of the if statement and predicate is true and I cover the else branch of the if statement when the predicate is false.

So, for this particular if statement: suppose I set x to be equal to 3 and the value of the variable flag to be equal to true then the whole predicate with this and operator evaluate to be true. So, this would exercise the then part of the if statement. And suppose I give a value of x to be 7 and let say I set the Boolean variable flag to be true, then this predicate would evaluate to be false, because this first condition will evaluate to be false. So, then this will mean that this whole if statement will have to exercise the else part of the code that is present after the if statement.


So, I say this is my test requirement; my test requirement is to be able to achieve predicate coverage on this if statement and these are the two test cases that achieve this test requirement. We will see how to define such coverage criteria based on graph, based on predicates, based on sets of inputs and how to automatically design test cases that will tell you whether the particular coverage criteria is achieved or not, and if it is achieved how is it achieved.

(Refer Slide Time: 24:21)



The slide is titled "Test coverage criteria" in a blue header. It contains a bulleted list of definitions and a logo in the bottom left corner.

- **Test requirement:** A specific element of a software artifact that a test case must meet/satisfy or cover.
- **Coverage Criterion:** A rule or collection of rules that impose (test) requirements on a set of test cases.
- Given a set of test requirements TR for a coverage criterion C , a set of test cases T **satisfies** C iff
For every test requirement $tr \in TR$, there is at least one test t in T such that t satisfies tr .
- A particular coverage criteria may or may not be satisfiable.
- Coverage criteria that cannot be satisfied are called **infeasible**.



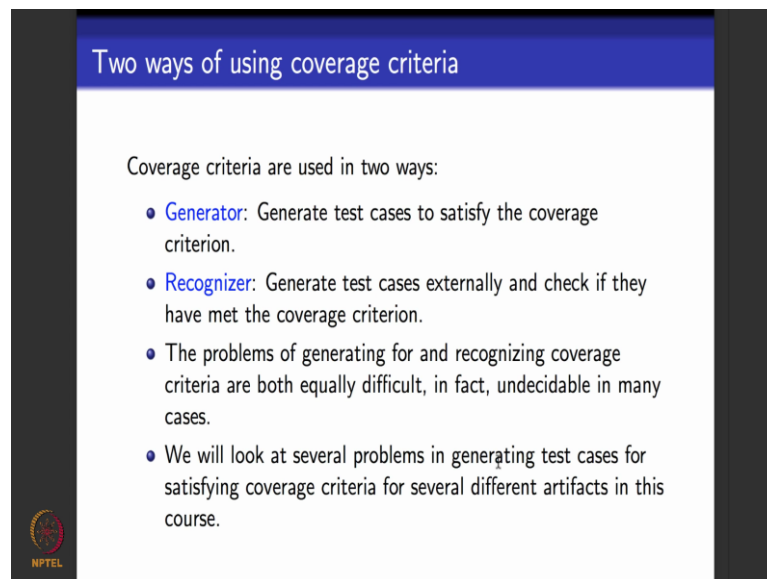
So, here are some definitions. What is the test requirement? A test requirement is basically a requirement that you say about a test case. You gave a thing like please design a test case to cover this if statement, to execute this if statement once for the then part once for the else part. You might say, please design a test case that will execute this while statement at least three times. You know these are all test requirements. They basically give you requirements on how to design test cases.

What is a criterion? A criterion is a rule or a set of rules that impose certain requirements on a set of test cases. Like in the previous slide if you say my test requirements is to test this if statement to be true once and to be false once then the criteria is what we call as predicate coverage. So, given a set of test requirements for a coverage criteria C , we say that a set of test cases satisfies the coverage criteria C if every test case in that set of test requirements satisfies the coverage criteria C ; there is nothing more to it.

So, it is to be noted that a particular coverage criteria may or may not be feasible. Like for example: you take an if statement that occurs in something like this, let us say this if statement is labelled by a predicate that can never be made false. For example right, it could be labelled by a predicate that is always true or a tautology. In which case I say that predicate coverage criteria on this if statement becomes infeasible, because predicate coverage says that you make this predicate that labels an if statement true once and false once.

The predicate is such that it can never be made false. If it can never be made false then I cannot achieve predicate coverage. When I cannot achieve predicate coverage then I say that the particular coverage criteria that I cannot achieve are what is called infeasible. So, it is undecidable problem to check or an arbitrary coverage criterion whether it is feasible or not. What we will see several heuristic or techniques that will let you decide implicitly whether particular coverage criteria is feasible or not; and if it is feasible to be able to design test cases that will satisfy the coverage criteria.

(Refer Slide Time: 26:39)



Two ways of using coverage criteria

Coverage criteria are used in two ways:

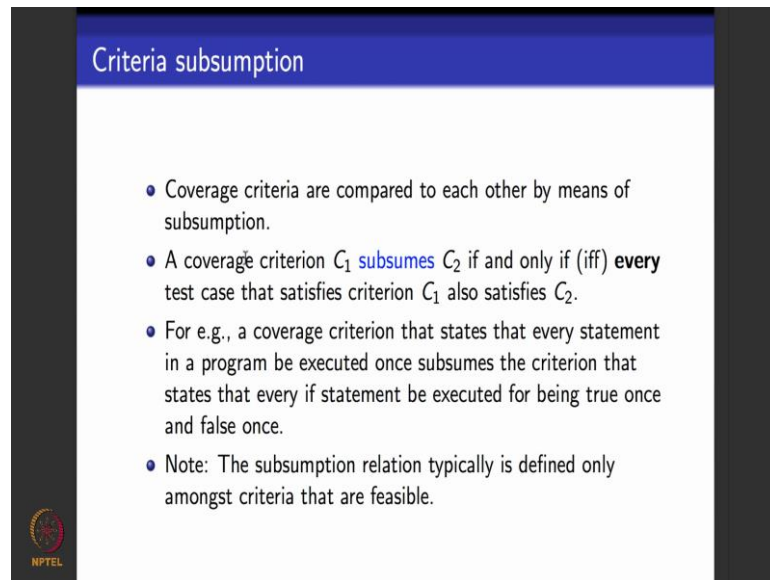
- **Generator:** Generate test cases to satisfy the coverage criterion.
- **Recognizer:** Generate test cases externally and check if they have met the coverage criterion.
- The problems of generating for and recognizing coverage criteria are both equally difficult, in fact, undecidable in many cases.
- We will look at several problems in generating test cases for satisfying coverage criteria for several different artifacts in this course.

NPTEL

So, how do we design test cases that satisfy the coverage criteria? There are two base to do it: one is to automatically generate test cases that satisfy the coverage criteria, and the next one is to generate test cases externally; externally means you generated arbitrarily and then somebody gives you the coverage criteria. Now you take this set of generate test cases and the coverage criteria and you check if these test cases meet those coverage criteria so that is done by what is called a recognizer. And directly given a coverage criteria generating test cases for that coverage criteria is done by what is called generator.

Both generator and recognizer for testing based on coverage criteria are in their most generality, undecidable problems. In fact several times the algorithms for decidable fragments also have high complexity, but we will look at algorithms, nonetheless, without worrying about what the complexities and see how to use them to be able to generate test cases.

(Refer Slide Time: 27:47)



The slide is titled "Criteria subsumption" in a blue header bar. Below the header, there is a white area containing a bulleted list of four points. In the bottom left corner of the slide, there is a small circular logo with the text "NPTEL" underneath it.

- Coverage criteria are compared to each other by means of subsumption.
- A coverage criterion C_1 **subsumes** C_2 if and only if (iff) **every** test case that satisfies criterion C_1 also satisfies C_2 .
- For e.g., a coverage criterion that states that every statement in a program be executed once subsumes the criterion that states that every if statement be executed for being true once and false once.
- Note: The subsumption relation typically is defined only amongst criteria that are feasible.

Now, suppose I have a particular thing and I define several different coverage criteria on a particular test requirement. So, I should know how do each of these coverage criteria compare to each other. Like for example, I in a piece of code I could have two different coverage criteria: one coverage criterion says that- you design a set of test cases that will execute every statement once. Another coverage criterion says that- you design a set of test cases which will execute every branch for the then part and for the else part once.

So, how do I know which of this is better and the work that I do for achieving one coverage criteria, implicitly also meets another coverage criteria. So, the notion of subsumption comes to our rescue here. So, we say that a particular coverage criteria C_1 subsumes another coverage criteria C_2 if every test case that satisfies criterion C_1 also satisfies criterion C_2 . Like for example, suppose I told you there was a criteria with says executive every statement once right, which means executive every statement in the program. And let us say there was another other criteria which says that is executed in every branch that you encounter, execute the true part executive the false part.

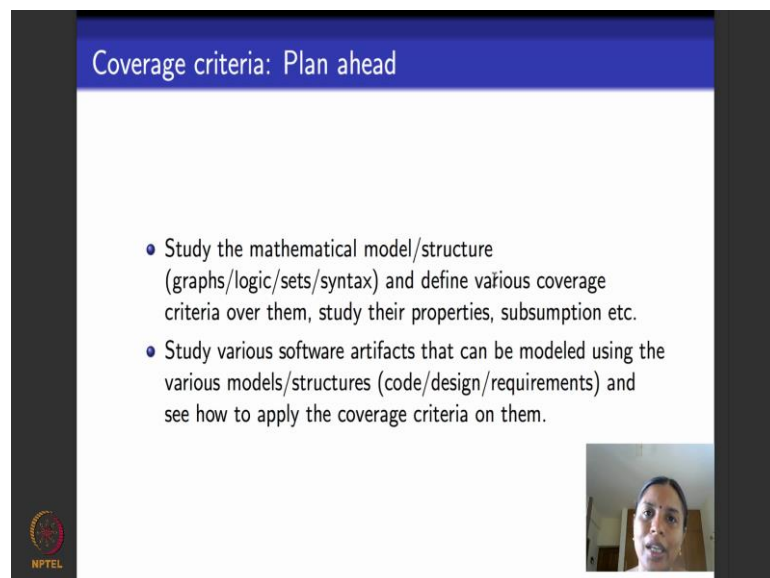
It is obvious that the first coverage criteria with says executive every statement one subsumes the second coverage criteria, because if I execute every statement once I would also be executing the statements of the then branch and the statements of the else branch.

So, this is how I use the notion of subsumption to be able to compare coverage criteria. It is important to be able to compare coverage criteria to reduce the burden on you

repeatedly generating test cases for different coverage criteria that are actually subsumed by the other coverage criteria. So, it is important to know if I generate test cases for one kind of coverage criteria what other parts of other different coverage criteria I have already achieved because of this. So, the notion of subsumption will help you to answer these kinds of questions.

One important note to observe that is that suppose a particular coverage criteria is infeasible then I really do not look at what subsumption for that coverage criteria, I define subsumption only for feasible coverage criteria.

(Refer Slide Time: 30:14)



The slide has a blue header with the text "Coverage criteria: Plan ahead". Below the header, there are two bullet points:

- Study the mathematical model/structure (graphs/logic/sets/syntax) and define various coverage criteria over them, study their properties, subsumption etc.
- Study various software artifacts that can be modeled using the various models/structures (code/design/requirements) and see how to apply the coverage criteria on them.

In the bottom right corner, there is a small video inset showing a person speaking. In the bottom left corner, there is a small NPTEL logo.

So, what is the plan ahead for the next weeks? What we will do is that we will consider software as various mathematical models or structures. So, what are the four structures that I said we will look at? We will look at graphs as model software artifacts, we look at logical predicate that occur in software artifacts, we will look at sets of inputs and outputs that are given to the software, and finally we will look at the underlining grammar or the syntax of software.

So, each of these models as structures we will define coverage criteria and discuss algorithms that will let us you design test cases to achieve coverage criteria. Of course, we look at subsumptions of coverage criteria and so on and so forth. So, what we will do first is we look at the model at an abstract level. Just as a graph, as a predicate, and then define algorithms for coverage criteria. Post that we will look at various software

artifacts; so we will see how to take code and model it as a graph, and what are the coverage criteria that we look at the graphs and how do those help to test various aspects of the code. Then we will take design and model it as a graph, and then we will see what are the coverage criteria that we looked at for graphs that will be relevant to design test cases for these designs.

So, similarly when we go to logical predicates, we look at logical predicates as they occur in code, look at coverage criteria and then see how these coverage criteria means testing which parts of the code. So, we look at these mathematical structures look at coverage criteria separately, then we look at how to model software artifacts using these structures and what do the various define coverage criteria mean for these structures.

So, next week when I begin my first module we will look at graph as models of software artifacts and define coverage criteria based on graphs so that will be the next lecture.

Thank you.

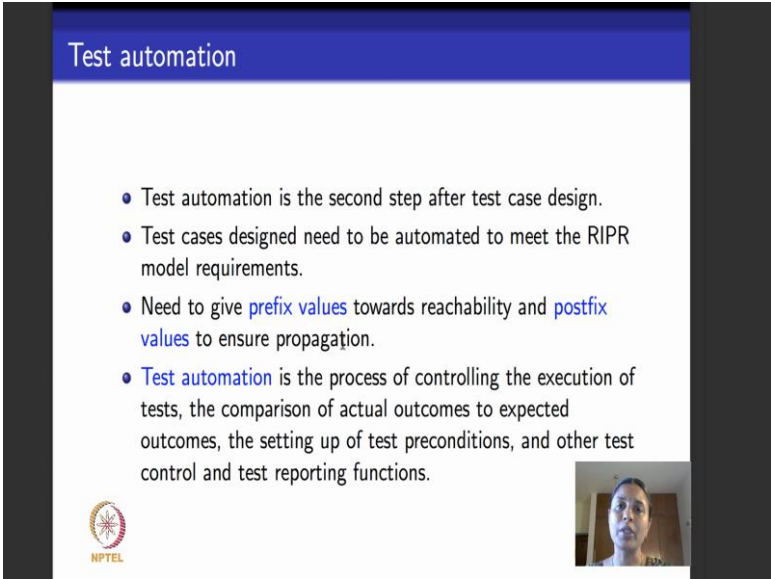
Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 04
Software Test Automation: JUnit as an example

Hello everyone. We will do the last module of the first week. In today's module I would like to concentrate on the second step of testing. If you remember in the previous modules we saw that there were four steps involved in software testing: your design test cases, then you automate test case design- that is you make the test design test case ready for execution, followed by the third step which is the test execution process, and finally when the test execution results are recorded you evaluate the test.

So, the second step in these four processes is that of test case automation. And the focus of this lecture is to understand what is test case automation and what exactly goes into it.

(Refer Slide Time: 00:54)



The slide is titled "Test automation" in a blue header. It contains a list of four bullet points:

- Test automation is the second step after test case design.
- Test cases designed need to be automated to meet the RIPR model requirements.
- Need to give **prefix values** towards reachability and **postfix values** to ensure propagation.
- **Test automation** is the process of controlling the execution of tests, the comparison of actual outcomes to expected outcomes, the setting up of test preconditions, and other test control and test reporting functions.

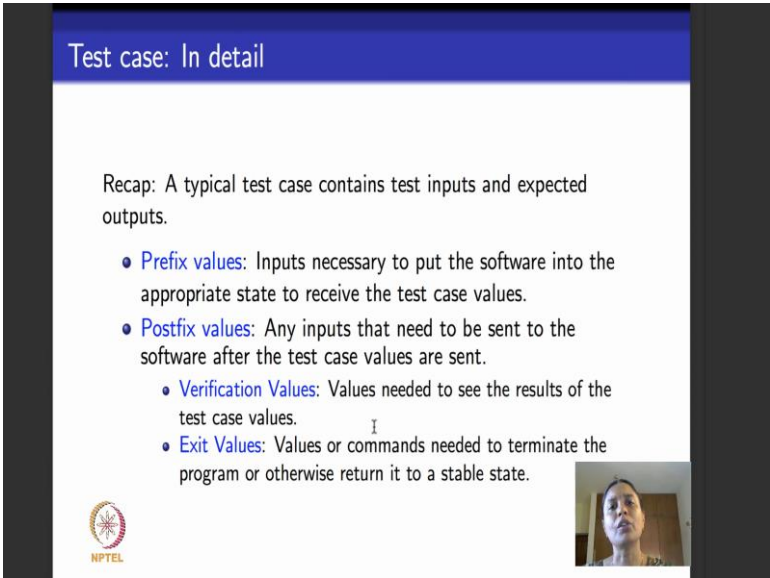
In the bottom left corner, there is a small circular logo with the text "IITB" and "HPTCL" below it. In the bottom right corner, there is a small video inset showing a man speaking.

In the previous module we saw that testability of a software, in turn, translates into observability and controllability, and these are measured using the RIPR model; R for reachability, I for infection, P for propagation and the final for R for revealing. So, how do I ensure that reachability propagation and revealing are done? I ensure that these three are done in the step of test case automation which is what we will look at in this module. So, how do we do that? We have to give what are called prefix values to ensure

reachability of a particular piece of code, and then once that test case design exercises that piece of code we have to give postfix values to the test case design to ensure that if there is an error that gets propagated outside.

So, all these things are what are called test automation. To formally define test automation it is the process of controlling the execution of tests, and actually ensuring the reachability is done by giving prefix values of preconditions. Then, you execute the test and compare the actual out come to the expected outcome and then you know report the results of your test case execution.

(Refer Slide Time: 02:11)



The slide is titled "Test case: In detail" in a blue header. Below the header, it says "Recap: A typical test case contains test inputs and expected outputs." followed by a bulleted list:

- **Prefix values:** Inputs necessary to put the software into the appropriate state to receive the test case values.
- **Postfix values:** Any inputs that need to be sent to the software after the test case values are sent.
 - **Verification Values:** Values needed to see the results of the test case values.
 - **Exit Values:** Values or commands needed to terminate the program or otherwise return it to a stable state.

In the bottom left corner is the NPTEL logo, and in the bottom right corner is a small video inset showing a person's face.

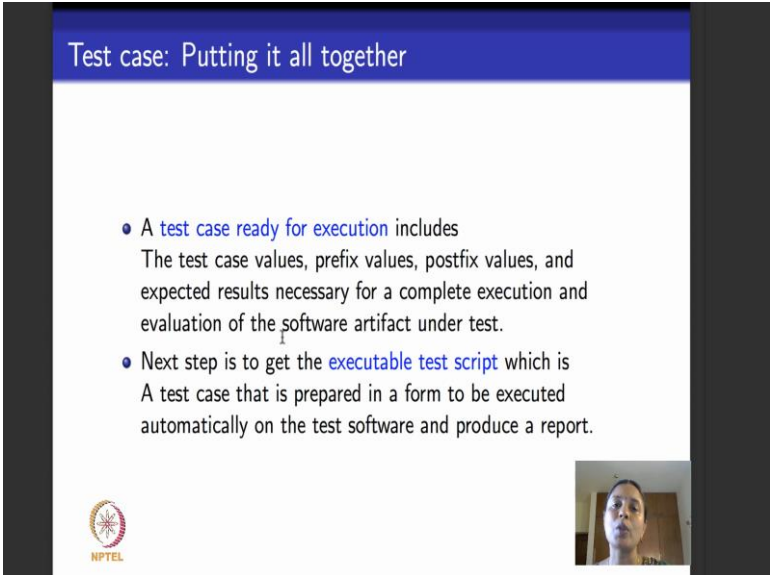
So, we will recap what a test cases and then see what are the add ones that we have to do to a test case to make it ready as an executable test script. If you remember from the first lecture what was the test case; test case basically contains test inputs and expected out puts. And if the expected outputs match the actual output after execution then you say that the test cases passed otherwise it is failed. This is a raw test case that has been designed.

Now, to make it into an executable test script the process of test automation has to add prefix values and postfix values to this test case. So, what are prefix values? They are basically inputs that are necessary to put the software or the software artifact into an appropriate state so has to be able to receive the actual test case for execution. And after the test case is executed, postfix values come into picture. What a postfix values? They

have values that are necessary so that the results of execution are sent to the software as an observable value by an external user.

Postfix values intern bifurcated into two categories: verification values and exit values. What are verification values? Verification values basically tell you that an exception has occurred this test case has passed or it has failed and it is a value that clearly tells you what is the result of test case execution. And what are exit values? Exit values are basically values or pieces of code that are needed to make sure that after the test case is executive the code appropriately finishes its execution fully and exits so, as to retain and reveal the error state, if there was any present during execution.

(Refer Slide Time: 04:04)

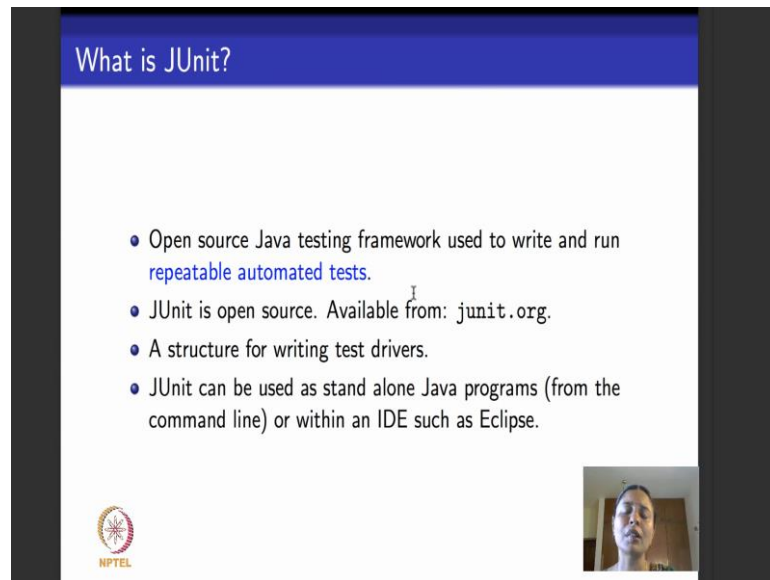


The slide has a blue header with the text "Test case: Putting it all together". Below the header, there are two bullet points. The first bullet point is "A test case ready for execution includes" followed by "The test case values, prefix values, postfix values, and expected results necessary for a complete execution and evaluation of the software artifact under test." The second bullet point is "Next step is to get the executable test script which is" followed by "A test case that is prepared in a form to be executed automatically on the test software and produce a report." In the bottom left corner, there is a logo for NPTEL. In the bottom right corner, there is a small video inset showing a person's face.

- A test case ready for execution includes
The test case values, prefix values, postfix values, and expected results necessary for a complete execution and evaluation of the software artifact under test.
- Next step is to get the executable test script which is
A test case that is prepared in a form to be executed automatically on the test software and produce a report.

So, now if you take the raw test case as it was designed and make it ready as a test script to be executed what are the summary of things that it has. It has the actual test case values, the prefix values, the postfix values, and as I told you it also has the expected outputs. So, when we put it all together and get it ready, a final result at the end of the test automation step should be to be able to get an executable test script, which is a test script that contains all these values and can be executed directly on the piece of code that it to being tested on.

(Refer Slide Time: 04:45)



What is JUnit?

- Open source Java testing framework used to write and run repeatable automated tests.
- JUnit is open source. Available from: junit.org.
- A structure for writing test drivers.
- JUnit can be used as stand alone Java programs (from the command line) or within an IDE such as Eclipse.

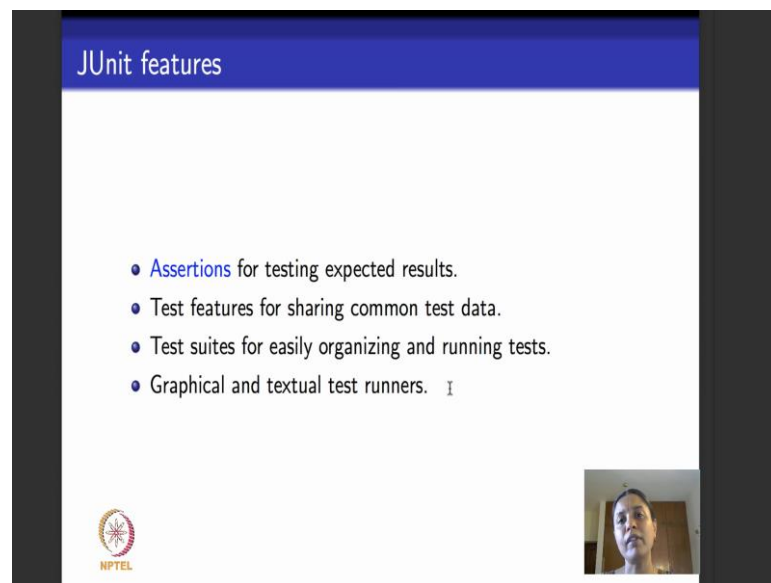
NPTEL

So, I will explain in detail how to give prefix values, how to give postfix values, and what happens in the process of test automation by using this open source tool; very good open source tool called JUnit as an example. Later in the part of the course when we see several examples of Java programs or C programs you could use the JUnit framework to be able to experiment with the test cases that you have designed as a part of assignments or other lectures that we will see in the course.

This module is not meant to be an exhaustive introduction of JUnit, but it is more meant to be used to interpret JUnit as a test automation tool, as a framework to understand how test automation works. In that process you will also learn some of the commands of JUnit which will be useful when you run your own experiments with JUnit at later points in the course. So, what is JUnit? It is an open source testing tool very popularly used in the industry. You can download it from [Junit.org](http://junit.org). And what are the things that it supports? It is very good for writing and executing test scripts.

So, once you design a test case you can automate it using JUnit and you can execute it and evaluate the results also using JUnit. It can be used as a standalone entity on Java programs or it can be used within an IDE like Eclipse.

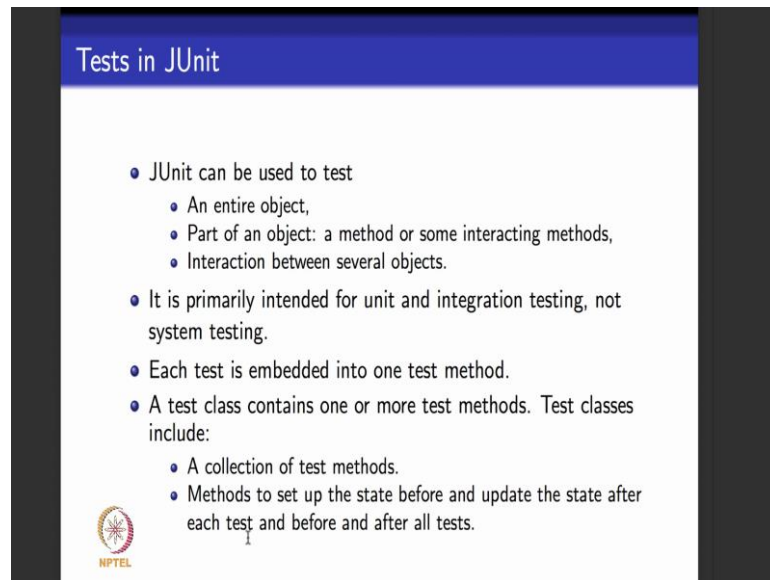
(Refer Slide Time: 06:13)



So, what are the high level features of JUnit? JUnit supports what are called assertions. How are assertions used? Assertions are basically statements that will always return true or false. You can view an assertion as if it returns true then everything is fine, the test case is passed. If an assertion returns false then there is an error that has been found and you can use this assertion to be able to reveal the error to the tester. So, we will assertions, how to use assertions through examples in JUnit.

JUnit also has test features for sharing common test data. Let us say two people are writing a common piece of code and they want to be able to share the test cases that they are writing then you can use JUnit to be able to do that. And JUnit also has suites for easily organizing, running, executing and observing tests. And of course, like many other tools it has a textual interface and it also has graphical interface.

(Refer Slide Time: 07:11)



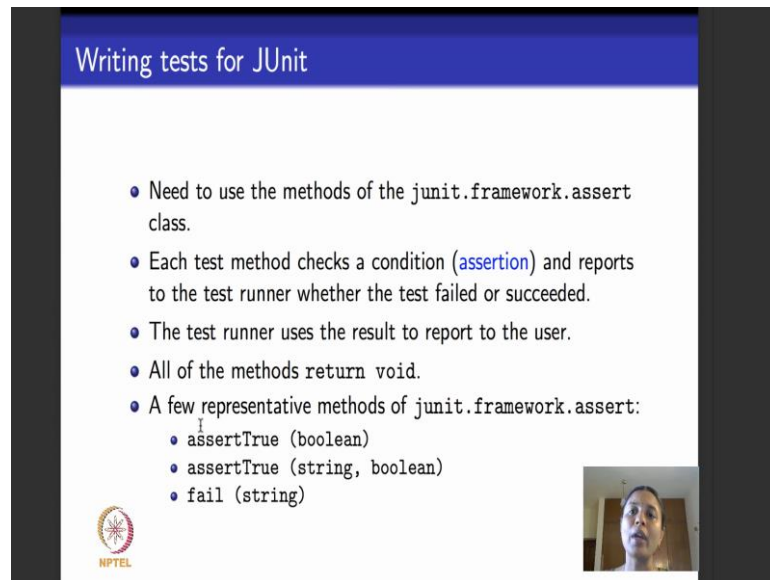
The slide is titled "Tests in JUnit" in a blue header. It contains a bulleted list of points about JUnit. The points are: JUnit can be used to test (An entire object, Part of an object: a method or some interacting methods, Interaction between several objects); It is primarily intended for unit and integration testing, not system testing; Each test is embedded into one test method; A test class contains one or more test methods. Test classes include: (A collection of test methods, Methods to set up the state before and update the state after each test and before and after all tests). There is an NPTEL logo in the bottom left corner of the slide content area.

- JUnit can be used to test
 - An entire object,
 - Part of an object: a method or some interacting methods,
 - Interaction between several objects.
- It is primarily intended for unit and integration testing, not system testing.
- Each test is embedded into one test method.
- A test class contains one or more test methods. Test classes include:
 - A collection of test methods.
 - Methods to set up the state before and update the state after each test and before and after all tests.

So, what can be JUnit be used for testing? The main thing is JUnit used for testing Java programs. So, it can be used as a very good unit testing tool or it can be used as a very good integration testing tool. It obviously cannot be used as a system testing tool. In system testing if you remember what I had told you, we take the inputs put the system is a part of the input server, connected to the database and let the outputs be observed as commands and so on.

JUnit being a testing tool for Java cannot be used for system testing, but can very well be used for unit testing and integration testing phases. So, it can be used to test an entire object, it can be used to choose just one method a certain interacting methods within an object it is up to you. So, JUnit has what are called test methods and each test is embedded into one test method, then it has a test class that contains one or more test methods. Test class, apart from this, can also have method that I used to setup the software to the state before and update the state after each test. So, they are these methods can also be used to do the prefix and postfix values that I had telling you about. And then there are code test methods which actually contain the test cases that have to be executed.

(Refer Slide Time: 08:34)



Writing tests for JUnit

- Need to use the methods of the `junit.framework.assert` class.
- Each test method checks a condition (**assertion**) and reports to the test runner whether the test failed or succeeded.
- The test runner uses the result to report to the user.
- All of the methods return `void`.
- A few representative methods of `junit.framework.assert`:
 - `assertTrue (boolean)`
 - `assertTrue (string, boolean)`
 - `fail (string)`

NPTEL

NPTEL

So, how do I write tests for JUnit? The first thing that we will understand while writing test using JUnit is to be able to use assertions. As I told you a little while ago what is assertion used for; assert is like a debug but in the context of testing. So, I say assert something if this test case passes, assert something if this test case fails. Assertion is always a Boolean expression. The value inside an assert always evaluate to true or false.

The implicit understanding while writing assertions is that if it evaluates to true then everything is fine your test cases passed so you can move on. But if an assertion evaluates to false then it indicates or it might indicate an error state. So, you will typically put a print statement there, saying what is gone wrong and you throw an exception or you print an appropriate warning and so on. So each test method that is used inside JUnit basically checks for a condition which is nothing but the assertion. And reports to the main test runner method about whether the test is passed or failed.

The test runner method uses the result of this assertion failing of passing to be able to report the result to the end user. Here are some examples of how asserts run. So, it is present in JUnit or framework or assert. So, you can do something like

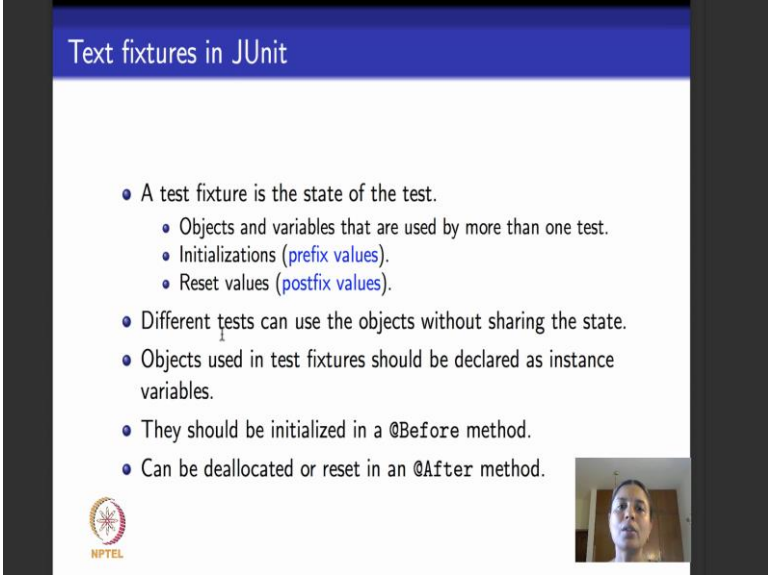
`assert true Boolean.`

So, this just says that if this Boolean predicate inside this asserts true evaluates to true then you just quietly say that it is true. If it evaluates to false maybe he will give a warning and terminate if it require.

The second asserts true test two arguments it takes a string and it takes a Boolean predicate. The idea here is that if the Boolean predicate evaluates to be true then you do not do anything. And if the Boolean predicate evaluates to be false then you print the string. So, it can be used as a way of warning the user.

A third assertion is what is called fail assertion which basically says only if it fails, if a particular thing fails then you output string as a warning to the user. So, when we see examples of code that we write with JUnit, I will show you examples of how to use all these assertions.

(Refer Slide Time: 10:58)



The slide is titled "Text fixtures in JUnit" in a blue header. It contains a list of bullet points explaining test fixtures. In the bottom left corner is the NPTEL logo, and in the bottom right corner is a small video feed of a person.

- A test fixture is the state of the test.
 - Objects and variables that are used by more than one test.
 - Initializations (*prefix values*).
 - Reset values (*postfix values*).
- Different tests can use the objects without sharing the state.
- Objects used in test fixtures should be declared as instance variables.
- They should be initialized in a `@Before` method.
- Can be deallocated or reset in an `@After` method.

So, now how to write a test fixture? Assertion is a core component which tells you when a test cases passed or failed. Now we have to write still prefix values, postfix values, how do you do that. So, how do I do that? Prefix values are initialized in what is called before method; 'at before' method and postfix values are given in a method called 'at after' method- the names are very intuitive and very easy to remember.

(Refer Slide Time: 11:27)

A simple example

Consider the code for addition below:

```
public class Calc
{
    static public int add(int a, int b);
    {
        return a+b;
    }
}
```

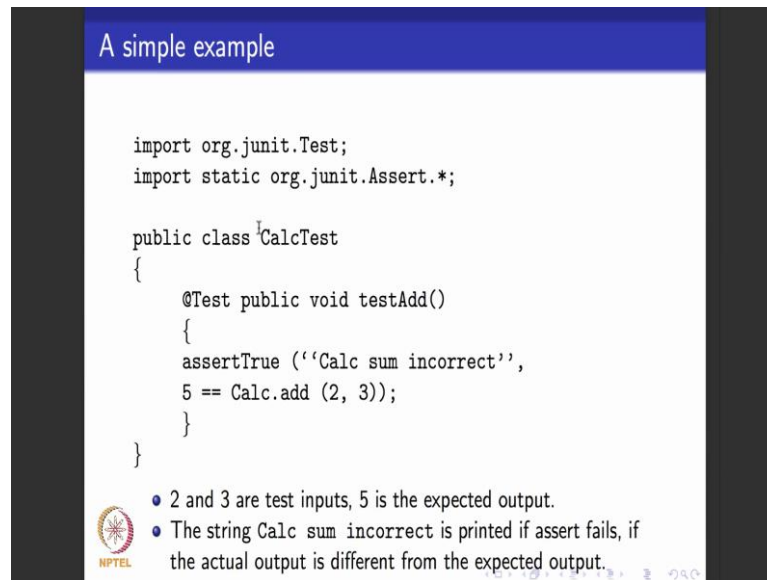
NPTEL

So, I will first walk you through a couple of examples. We will start with the simple code for addition as an example, then I will look at another example which returns the minimum element in the list, and tell you how to write prefix values, how to write postfix values, how to give the test cases, and how to write assertions that will output the result of the test case passing or failing.

So, we will start with an example of a code that does addition. So, here is an example of a code that does addition there is nothing complicated to it, it takes two integers a and b and it returns the sum of a and b which is a plus b. Now suppose you are given the task of testing this code. So, let us recap and understand what a test case is.

So, test case should give inputs and it should give expected outputs. What are inputs to this piece of code? Inputs are one value for a and one value for b, and expected output is the actual sum of a and b. Now what you have to do is a part of the automation you say that you please take these inputs and compare it to the expected output. If the actual output matches the expected output fine, everything is working fine, but if the actual output differs from the expected output then you use assertions to be able to flag the fact that the test case execution has failed.

(Refer Slide Time: 12:50)



A simple example

```
import org.junit.Test;
import static org.junit.Assert.*;

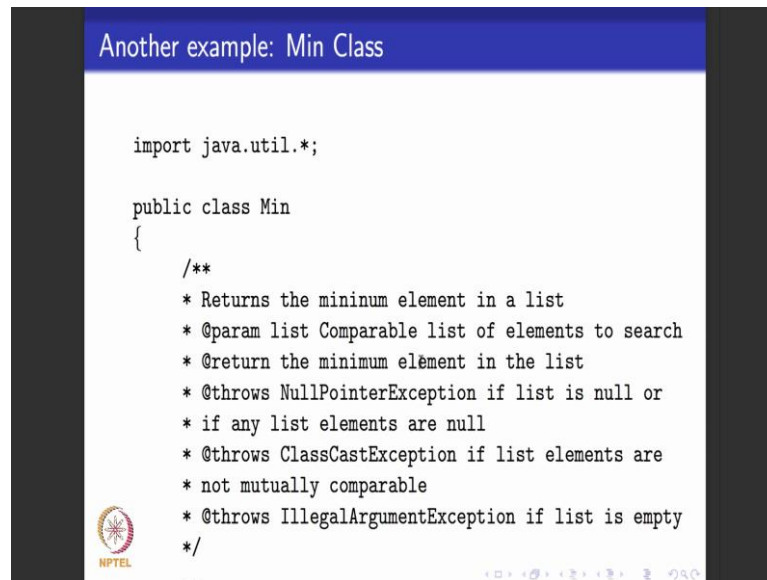
public class CalcTest
{
    @Test public void testAdd()
    {
        assertTrue ("Calc sum incorrect",
            5 == Calc.add (2, 3));
    }
}
```

- 2 and 3 are test inputs, 5 is the expected output.
- The string Calc sum incorrect is printed if assert fails, if the actual output is different from the expected output.

So, how will you do it in JUnit? This is how you do it in JUnit. So, these has standard things that you have to import, every time you will write any program any test method using JUnit for test case automation and execution. So, you import what is called JUnit or test and you import this library of assertions. So, I am writing something called the test for the calc method. How do I do? I write these two statements the first statement says that- you assert true and it gives a string, and the second statement actually runs the addition code that we saw in this slide with two actual test inputs--- 2 and 3. And if 2 and 3 are the test inputs assuming that addition works correct what is the expected output? The expected output should be 5. So, it compares it to 5.

So, what it says is that you run this add program on inputs 2 and 3 and check if the result that is outputted by the add program on these inputs is actually equal to 5. If it is actually equal to 5 then you exit, test cases passed. But if it is not equal to 5, if the calc program actually has an error; in this case it does not have an error because it just a simple program that returns a plus b. But assuming that it does have an error you write it for some other program, then this assert true will take over and it will output this string. So, it will say that calc sum is incorrect. So, it will say that there is a error somewhere in the code.

(Refer Slide Time: 14:22)



So, now we look at another example, slightly longer than that simple toy addition example. So, this is an example of a piece of code, Java code, that written the minimal element in a list. So, this code is very well debugged; in the sense that it takes care of all exception conditions, it takes care of what if the list is empty there is nothing to written, what if the list has entities that cannot be compared at all. For example, suppose the list has a string and a number, the list has a string and a Boolean value--- there of different types they cannot be compared.

So, in all these cases this code is meant through several different kinds of exceptions. So, these are what are called a part of debugging. Whether developer himself takes care of all the corner cases like wrong inputs, wrong data types etcetera which need not be taken care of by a tester. So, this code is well written to be able to take care of all the exceptional cases. Of course, the tester's job would be able to test for all these features also as we will see through examples, but we will first see what the code does and what are the various exceptions that it takes care of.

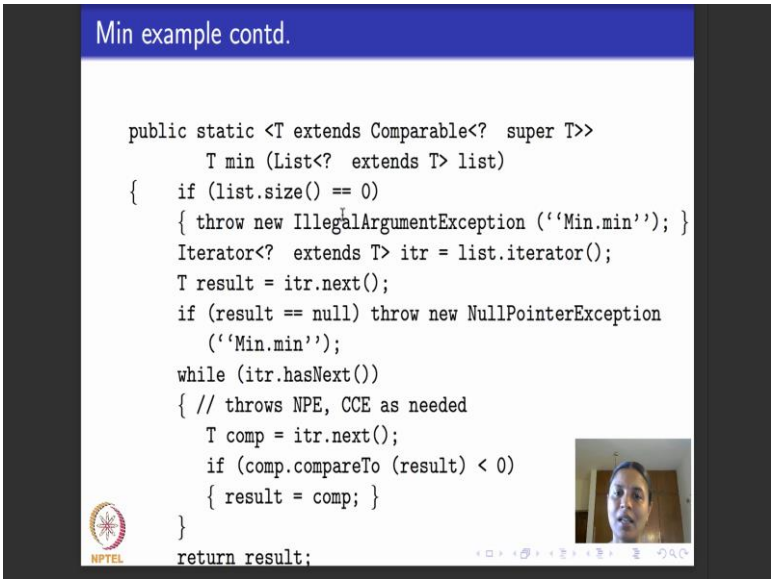
So, this class is called min and as I told you it returns the minimum element in a list and it takes as input comparable list of elements. What I said is that all of them should be comparable, we should all be a list of numbers or they should be a list of strings that I can compare them with respect to the lexicographic ordering. They should not be

incomparable types. And what is the main functionality of this class min? It supposed to return the minimum element in the list.

As I told you this class min is also supposed to take care of exceptions, like wrong inputs, incomparable inputs and so on. So, it has several exceptions. The first exception that it has is what is called a null pointer exception. It throws the null pointer exception if the list is empty or if any of the elements of the list are empty. And the second kind of exception that it throws is what is called as ClassCastException which it throws if the list elements are not comparable to each other. As I told you one is a string and the other is a Boolean constant how do you compare them, so you have to throw an exception. It also throws an illegal argument exception if the list that is passed to it is empty, there are no elements to compare.

So, I have split this code across two slides because otherwise it would not be readable. So, we will move on and look at the rest of the code in the next slide.

(Refer Slide Time: 16:54)



Min example contd.

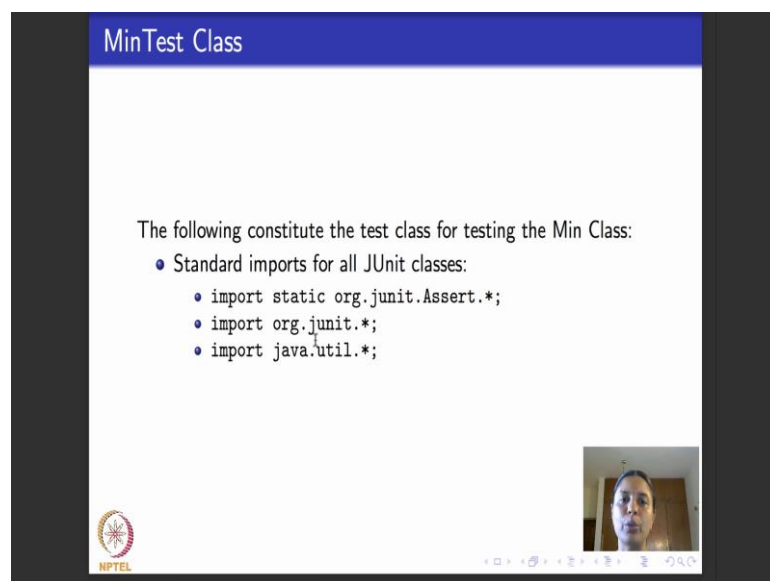
```
public static <T extends Comparable<? super T>>
    T min (List<? extends T> list)
{
    if (list.size() == 0)
    { throw new IllegalArgumentException ("Min.min"); }
    Iterator<? extends T> itr = list.iterator();
    T result = itr.next();
    if (result == null) throw new NullPointerException
        ("Min.min");
    while (itr.hasNext())
    { // throws NPE, CCE as needed
        T comp = itr.next();
        if (comp.compareTo (result) < 0)
        { result = comp; }
    }
    return result;
}
```

So, what is the main code look like? This is how the main code looks like, I just glanced through it because I do not want to read the code line by line, is just a standard written Java program that takes a list called T. And then if the list is empty then it throws an illegal argument exception otherwise it repeatedly compares it to the next element in the list and returns the result in this value called result; it returns the minimum element. So, this is a piece of Java code. So, just to recap what its main functionalities are, it takes a

list of comparable elements as its argument and then returns the minimum element in the list.

Suppose the list does not have comparable elements, the list is empty; the list has other kinds of problems this code does exception handling very well. So, now our job is to be able to design test cases for this minimum program and see how you can use the second step which is the test automation step that is the focus of this lecture to be able to test this program.

(Refer Slide Time: 18:02)



The slide is titled "MinTest Class" in a blue header. The main content area is white and contains the following text:

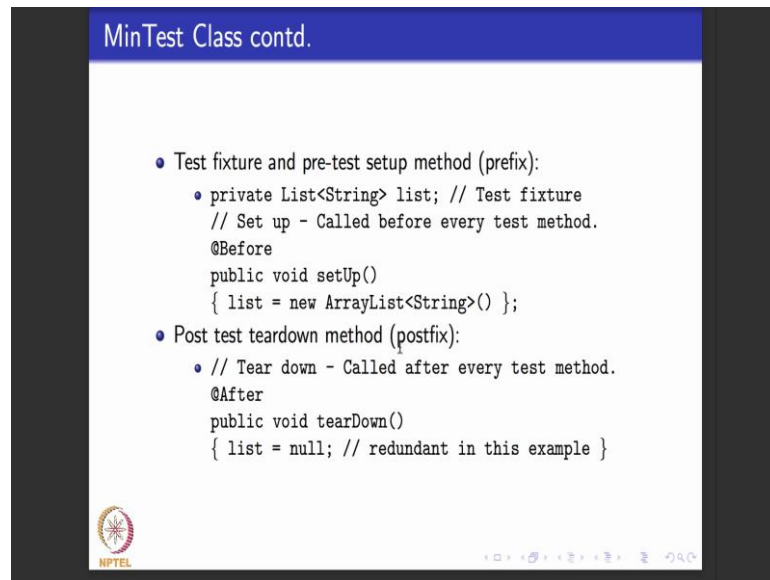
The following constitute the test class for testing the Min Class:

- Standard imports for all JUnit classes:
 - `import static org.junit.Assert.*;`
 - `import org.junit.*;`
 - `import java.util.*;`

In the bottom right corner, there is a small video feed of a person. In the bottom left corner, there is a logo for NPTEL.

So, how do I do? The first step as I told you is because we want to be able to use the tool JUnit we have to import all the standard classes; we have to import JUnit assert, we have to import JUnit you have to import the util library. After this the next job is to be able to give prefix value and postfix values to the code.

(Refer Slide Time: 18:25)



The slide is titled "MinTest Class contd." and contains the following code:

```
• Test fixture and pre-test setup method (prefix):
  • private List<String> list; // Test fixture
    // Set up - Called before every test method.
    @Before
    public void setUp()
    { list = new ArrayList<String>(); };

  • Post test teardown method (postfix):
    • // Tear down - Called after every test method.
      @After
      public void tearDown()
      { list = null; // redundant in this example }
```

The slide also features the NPTEL logo in the bottom left corner and navigation icons in the bottom right corner.

How is that done? Prefix values is given by this kind of, as I told you right, by this act before thing. So, what you do is that you give a test fixture and a pre test setup method. So, you have it like this--- you give it a list which is a test fixture and then you set it up which is called before the main test method. The main test method actually has the test cases for execution.

So, you say that this is the thing and I pass a new array for this. And after this in the main test case the main test method what we called containing the test cases, and post this I have what is called teardown method which gives the postfix values which basically in this case is not relevant because there is nothing much to do, but assuming that you had a fairly large piece of code postfix teardown method will actually do the rest of the execution to be able to see the output as being produced by the code. In this case because it is a small example we will directly see the output. So, there is not much postfix activity to be done here.

(Refer Slide Time: 19:29)

Min Test Cases: NullPointerException

A test case that uses the fail assertion.

```
@Test public void testForNullList()
{
    list = null;
    try {
        Min.min (list);
    } catch (NullPointerException e)
    {
        return;
    }
    fail (NullPointerException expected);
}
```

NPTEL

So, here is an example of a test case that uses the fail assertion. If you remember I had told you there are three kinds of assertions. So, if you give me a few seconds I will go back to that slide. Remember there were three kinds of assertions I used took different kinds of assert true which basically return a warning if this Boolean string that is passed to it. If this Boolean predicate that is passed it is returns false or it returns string if the Boolean predicate that is passed to it is return false. And then third kind of assertion will returned this string if it fails.

So, here for this min example we will use the fail assertion and here is a test case that uses the fail assertion. What is this test case method called it is called test for null list, it is a void method and it passes an empty list and then its tries to see if the code actually throws a null pointer exception. And if it does not throw a null point exception using this try and catch, it will use the fail assert to say that it had actually expected a null pointer exception which did not happen. So, there is an error in the code. In our code this will not come because this error is handled.

(Refer Slide Time: 20:49)

Min Test Cases: Empty element

This is for the special case of an empty element.

```
@Test (expected = NullPointerException.class)
public void testForSoloNullElement()
{
    list.add (null);
    Min.min (list);
}
```

The slide includes a small video inset of a person in the bottom right corner and an NPTEL logo in the bottom left corner.

So, here is another example of a test case that tests our min code for the special case of an empty element. So, what it says is that I tried to add an empty element in the list and I tried to compute its minimum. And in this case because the code is well written this is also taken care of, it will return find and there is no error in this code even for this kind of test case.

(Refer Slide Time: 21:20)

Other tests with JUnit

- **Data-driven tests:** Data-driven unit tests call a constructor for each collection of test values.
 - Same tests are then run on each set of data values.
 - Collection of data values defined by method tagged with @Parameters annotation.
- Helps to test a function with multiple test values.

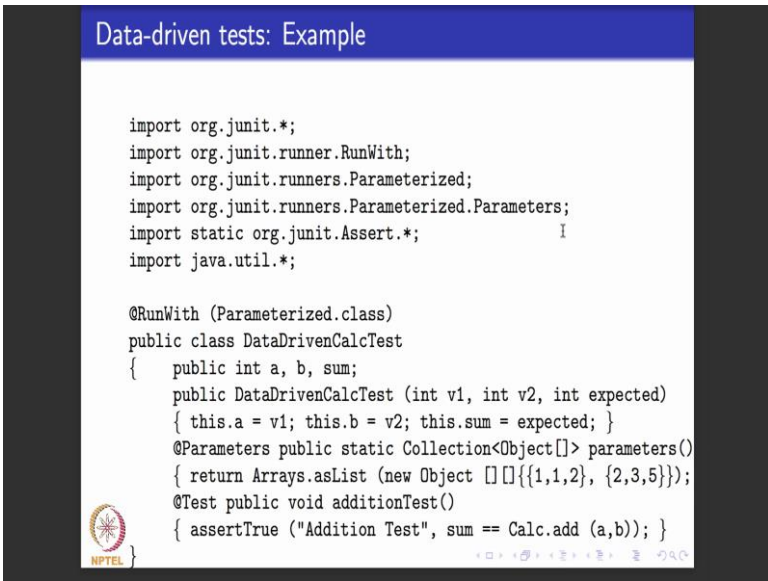
The slide includes a small video inset of a person in the bottom right corner and an NPTEL logo in the bottom left corner.

Now, what are the other things that we can do with JUnit. We saw two examples this is with testing for two exception cases. Of course, you can go on testing this minimum code

for several other exception cases, because the code handles several other an exception handling mechanism. But suppose I have to test the main functionality of the code. What is the main functionality of the code? The main functionality of the codes to be able to give a list of elements that are comparable and check if it actually returns the minimum value from the list; so for that I need to be able to give data, I need to be able to pass a list of compatible elements. How does that happen? How does one do it JUnit?

For giving data to test with JUnit we have a constructor for them. So, that constructor what it does is that you can passed several different data to it and the same tests are run for each set of data values and the collection of these data values are defined by a method tag with a @parameter. So, it basically helps to test a function with multiple text value. So, I can test a minimum function with several different lists and check for each of these lists does not return the minimum. I can check the add function that other toy example that we looked at with several different arguments a and b and check in each of these case does it actually return the sum of a and b.

(Refer Slide Time: 22:53)



```
import org.junit.*;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
import static org.junit.Assert.*;
import java.util.*;

@RunWith (Parameterized.class)
public class DataDrivenCalcTest
{
    public int a, b, sum;
    public DataDrivenCalcTest (int v1, int v2, int expected)
    { this.a = v1; this.b = v2; this.sum = expected; }
    @Parameters public static Collection<Object[]> parameters()
    { return Arrays.asList (new Object [] [] {{1,1,2}, {2,3,5}}); }
    @Test public void additionTest()
    { assertTrue ("Addition Test", sum == Calc.add (a,b)); }
}
```

So, I will go back to the add example and show you how to use this @parameters with the constructor to be able to provide data along with prefix, postfix and assert. So, here is the complete JUnit program for the same. So, import as I told you all these various classes, now what I do is I have to pass data. Now I am going back to the add examples. So, I have to always first two integers a and b to it and check if the sum of a and b is

actually returned by the addition code. So, I use this constructor and then I write this particular class that does the main testing.

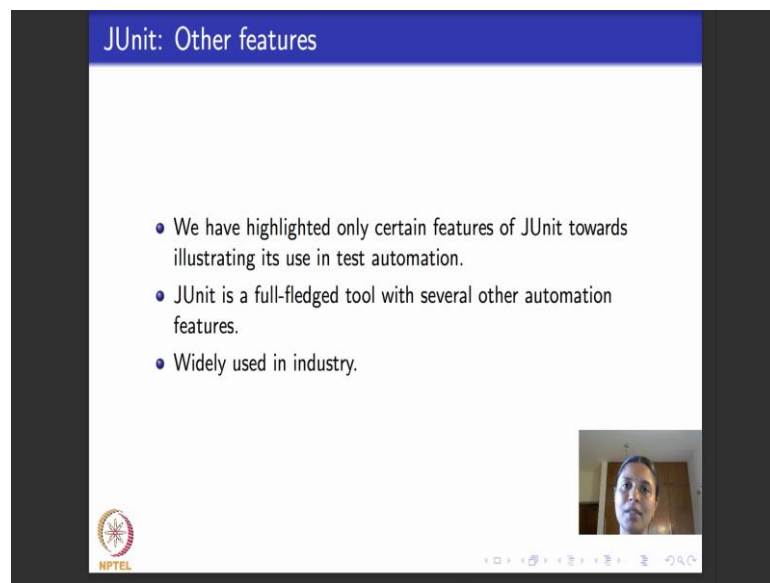
So, how does this work? If you see this dot a is equal to v 1 and this dot b is equal to v 2 these are the variables that are used to pass the actual parameters. And then the expected result is stored in this dot sum. And if you go here there is an array which the first value set of test case values that it passes are a is 1, b is 1, and expected result is 2. The second set of test cases that it passes are a is 2 b is 3 and the expected result is 5. In this example I have just given two so that I can explain it you, but you can pass an array of as many test cases as you want along with their expected outcome.

So, what it will do is; it will go back and execute this particular method. It will execute this particular method, and in any case for whichever test case if this assertion fails then it will output that this sum is incorrect. How it does is it will do it for each of these test case values and the expected output. And it will finally exit without giving any assertion violations if all of them pass. So, for our particular example the calculator addition was correct code so it will pass for all these examples.

So, similarly for the minimum element in a list also you can put it all together, right, check it for various null pointer, empty list and other kinds of exceptions, and after passing all these exceptions actually use this constructors class to be able to pass several different list of values and their minimum element and check whether the code actually returns the minimum element from this list; because that piece of test code in JUnit is fairly long to write it would not fit into even 2-3 slides, I have not given that as an example. What I can do is I will be putting it along with my notes feel free to look at it.

So, hopefully at the end of this exercise you would have understood how to give prefix values to a test case and how to actually give the test case values write asserts that will indicate whether the test case is passed or failed. And if needed, how to make postfix values in the codes such that the assert failure or assert pass actually reflexes output in the code.

(Refer Slide Time: 26:01)

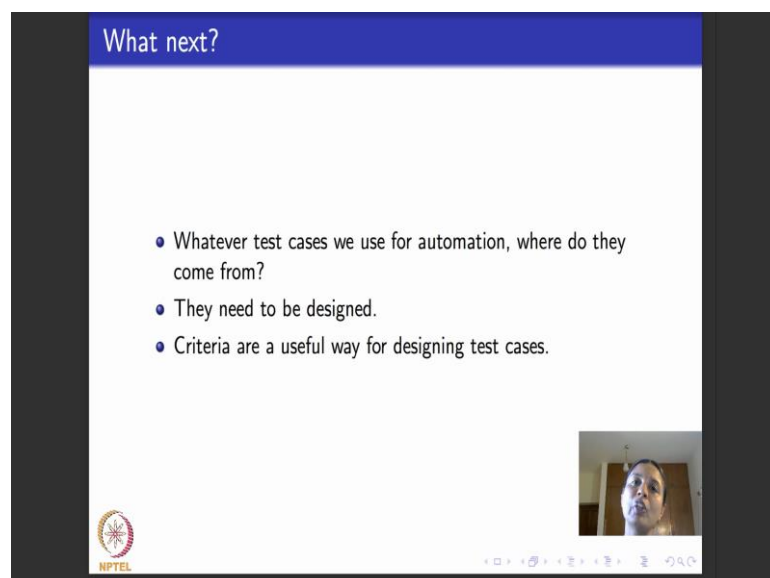


The slide has a blue header with the text "JUnit: Other features". The main content area is white and contains a bulleted list of three points. In the bottom right corner, there is a small video inset showing a man speaking. The NPTEL logo is in the bottom left corner, and navigation icons are in the bottom right corner.

- We have highlighted only certain features of JUnit towards illustrating its use in test automation.
- JUnit is a full-fledged tool with several other automation features.
- Widely used in industry.

To understand these we used JUnit. JUnit is a fairly extensive tool as I told you in the beginning; the purpose of this module was not to be able to teach you exhaustive features of the JUnit, but to be able to teach you how to use JUnit for test automation. So, feel free to go download JUnit and explore all its other features and try it out on your own Java programs to be able to see if you can test them or not.

(Refer Slide Time: 26:27)



The slide has a blue header with the text "What next?". The main content area is white and contains a bulleted list of three points. In the bottom right corner, there is a small video inset showing a man speaking. The NPTEL logo is in the bottom left corner, and navigation icons are in the bottom right corner.

- Whatever test cases we use for automation, where do they come from?
- They need to be designed.
- Criteria are a useful way for designing test cases.

So, where are we going on from this? So now, we saw how to automate and once you automate and execute JUnit also has an execution framework. As I told you it is fully

command driven, fully automated. So, there is nothing much to discuss about it, I will not be discussing about that in detail. After that you actually observed by using assertions the result of failures.

Now if you go back right to the first step; that is test case design it actually tells you what are the test cases that you pass on to a tool like JUnit for automating. How do you design test cases, how does one go about giving effective test cases without giving blind test cases and not hoping to find any errors. So, for this we go back to the problem of test case design. So, we will look at criteria based test case design, we model software using different mathematical model structure as I told you graphs, logical expressions, sets and so on. And teach you how to design test cases based on each of these. So, that will be the focus of my lectures beginning next week onwards.

Thank you.

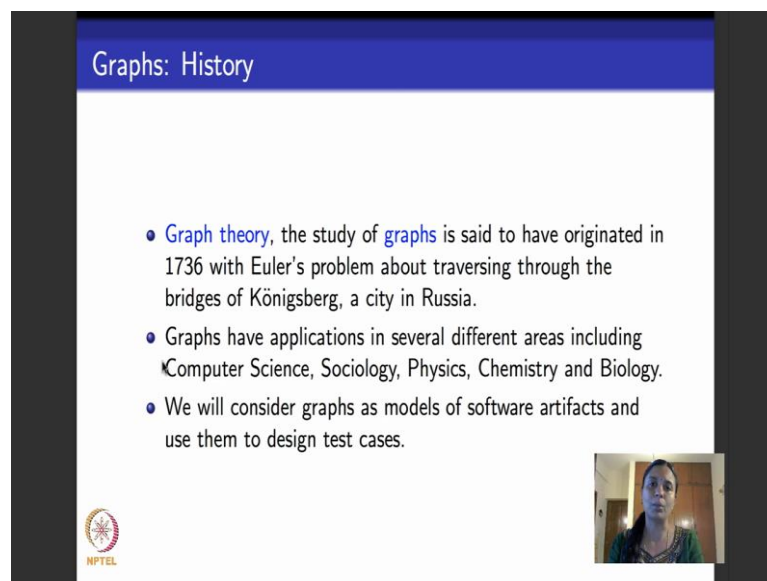
Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 05
Basics of graphs: As used in testing

Hello everyone. So, we begin looking at test case design algorithms in this module what I would be starting is to model software artifacts as graphs which will be one of the four structures that we would consider. If you remember, we said we would consider graphs, we would consider logic and expressions, we would consider sets that model inputs to software and finally, we would consider the underline grammar from which software programming language is written. So, we begin we are looking at test case algorithms that deal with graphs. So, why we look at test case algorithms that deal with graphs I would like to recap some basic terminologies related to graphs as we would be doing in this course in testing.

Graphs and graphs theory are vast areas, I will not be able to do just is to be able to cover even the basic minimum concepts that we deal with in graphs. So, we will restrict ourselves to just looking at terminologies that we need as far as test case design algorithms are concerned in this course.

(Refer Slide Time: 01:18)



The slide is titled "Graphs: History" in a blue header. It contains three bullet points: "Graph theory, the study of graphs is said to have originated in 1736 with Euler's problem about traversing through the bridges of Königsberg, a city in Russia.", "Graphs have applications in several different areas including Computer Science, Sociology, Physics, Chemistry and Biology.", and "We will consider graphs as models of software artifacts and use them to design test cases." In the bottom right corner, there is a small video inset showing a woman speaking. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.

- Graph theory, the study of graphs is said to have originated in 1736 with Euler's problem about traversing through the bridges of Königsberg, a city in Russia.
- Graphs have applications in several different areas including Computer Science, Sociology, Physics, Chemistry and Biology.
- We will consider graphs as models of software artifacts and use them to design test cases.

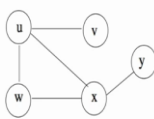
So, graph theory is a very old subject it is believed that study of graphs was initiated by Euler in the year 1736 when they were trying to look at this city called Königsberg in Russia and then they were trying to model a problem of crossing the bridges in this city in a particular way. So, they considered modeling this problem as a graph and graph theory is supposed to have originated with Euler's theorem which is considered an old theorem. So, you can imagine how old graphs is. Today graphs enjoy applications not only in computer science, but in several different areas all sciences physics, chemistry, biology, and in fact, it finds exist extensive applications in sociology where people look at social networks and other entities is very large graph models.

So, what do we do with graphs? We will consider graphs as models of software artifacts and see how to use graphs to design test cases as we want them.

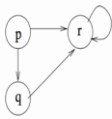
(Refer Slide Time: 02:19)

Graph

- A graph is a tuple $G = (V, E)$ where
 - V is a set of nodes/vertices.
 - $E \subseteq (V \times V)$ is a set of edges.
- Graphs can be directed or undirected.
- In an undirected graph, the pair of vertices constituting an edge is unordered, i.e., whenever $(u, v) \in E$ then $(v, u) \in E$ and vice versa.
- In a directed graph, the pair of vertices constituting an edge is ordered.



A simple undirected graph



A directed graph

So, we begin by introducing what a graph is say assume that you not seen it before. So, I will introduce you from the very basic concepts. If you seen it before feel free to, you know, sort of skip through these parts because they are talk about the basic terminologies related to graph. So, how does a graph look like? It looks like this. So, this is what is called an undirected graph which is simple in the sense that it does not have any self loops and here is what is called a directed graph, where the edges have directions.

So, graphs have two parts to it there are nodes or vertices. Sometimes I will use these terms synonymously interchangeably. Some people also call it is points and different

books might call them differently. So, that is typically sets that is marked using circles like this each circle is given a number. So, there are 5 vertices or nodes in this graph which are labeled as u, v, w, x and y. Similarly there are three nodes vertices in this graph labeled as p, q and r and graphs also have what are called edges. Edges are also called arcs or lines in certain other books. So, what is an edge? Edge is basically a pair of two nodes or two vertices. So, this pair of nodes or vertices can be ordered or unordered.

So, if the pair is unordered that is, I do not really worry about whether I am looking at the pair (u, v) or (v, u), then I say that the graph is an undirected graph and if the pair that I look at is ordered, like if I look at this figure of a directed graph on the right and looking at in ordered pair p comma r. So, there is an edge in this direction from p to r and in this graph there is no edge in the reverse direction right. So, such graphs are what are called directed graphs. Of course, it is to be noted that an edge can take vertex to itself there is no requirement that u should be different from v.

So, if you look at this directed graph the pair (r, r) constitutes this edge which is the self loop around the vertex r. So, when I look at a pair r comma r in an directed or in undirected graph because it is an reflects a pair I really do not worry about whether the pair is ordered or unordered right it does not matter. Otherwise, for each other pair of distinct vertices whether the pair is ordered or unordered defines the kind of graphs that we look at. Graphs could be directed as it is here or undirected as it is here.

(Refer Slide Time: 04:56)

Graphs

- Graphs can be finite or infinite. Finite (infinite) graphs have a finite (infinite) number of vertices.
- We will use finite graphs throughout our course.
- The **degree** of a vertex is the number of edges that are connected to it. Edges connected to a vertex are said to be **incident** on the vertex.

NPTEL

So, graphs can be finite or infinite. So, a finite graph typically has a finite number of vertices and infinite graph has an infinite number of vertices. Because our use of graphs in this course is to be able to use them as artifacts that modeled various pieces of software we will not really consider infinite graphs, we do not really have the need to modeled any kind of software artifact as an infinite graph. So, we will look at finite graphs throughout this course.

A few other terminologies in this graph. What is the degree of a vertex? The degree of a vertex gives you the number of edges that are connected to the vertex another term for connected that people use in the graph theory is to say that an edge is incident on the vertex. So, if we go back to the graph examples that we have in the previous slide if you take this vertex u in this undirected graph three edges are connected to this vertex you write one coming from w one connecting it to x and one connecting it to v . So, all these three edges are supposed to be connected to u or incident on u and so the degree of the vertex u is 3.

So, similarly degree of the vertex y is just 1 because there is only one edge that is incident on y . So, if you go to this directed graph the degree of the vertex r would be 3. So, when we count the degree of a vertex for an undirected graph we count the in-degree of the vertex, in-degree is the number of edges that come into a particular vertex. So, here there are three edges that come into r - one from p , one from q and one from r . So, we say r has in-degree 3, right. Similarly we also talk about an out-degree of a vertex in an directed graph. So, if it look at the vertex p , p has in-degree 0 because there is no edge that is coming into p , but p has out degree 2 because two edges go out of p .

If you look at the vertex q , q has in-degree 1 because this edge from p to q comes into q and q has out degree 1 because this edge from q to r goes out of q .

(Refer Slide Time: 07:16)

Initial and final nodes

- For many graphs, there are designated special vertices like **initial** and **final** vertices.
- These vertices indicate beginning and end of a property that the graph is modeling.
- Typically there is only one initial vertex, but, there could be several final vertices.
- Initial vertex represents the beginning of a computation (of a piece of code) and the computation ends in one of the final vertices.

A simple undirected graph A directed graph

So, moving on the graphs that we will look at will have several other things apart from just vertices and edges. So, one at a time we look at what are the add on or the additional features or annotations that we will consider to be a part of the graphs that we look at right. So, graphs could have special designated vertices called initial vertex and final vertex. So, what I have done here is I have taken the same graph that you saw two slides ago and marked the vertex u as an initial vertex u has an edge that is incoming line that is incoming into u , but it does not really have any vertex on the other side. So, such vertexes what is called an initial vertex or an initial node.

There could also be what are called final vertices. Final vertices are believed to be vertices that capture the end of some kind of computation in the models that we will look at when we will look at graph models corresponding to code and corresponding to design elements and become clear what is the purpose of final vertex, but from now you can understand it to be a final vertex is one in which computation is supposed to end in some way or the other and in our pictures that we will look at, final vertices will be marked by this concentric circle. So, if you see in this directed graph p is an initial vertex and r is a final vertex in this undirected graph u is an initial vertex and w is a final vertex.

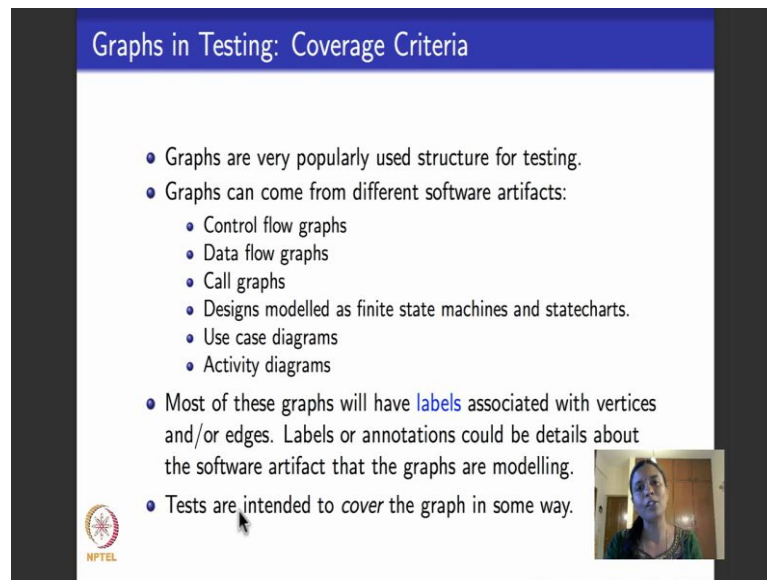
So, typically we believe that most of the software artifacts that we will consider like code mainly or design, is always supposed to be deterministic in the sense that its behaviour is definite and there is no non determinism in its behaviour. So, to be able to capture graph

as a model of a software that is meant to be deterministic, we all always say that the graph will typically always have only one initial state. If there were more than one initial states then it will be a bit confusing as far as determinism is concerned right because if there are more than one initial states let say there are three initial states where would you consider the computation as beginning from. You could interpret it as it is beginning from any one of the initial states, but then right there, the software is non deterministic right and we really do not typically look at nondeterministic software. Non deterministic implementations of software do not exist.

We always look at deterministic implementations of software and hence we will consider a graph models to always have only one initial state, but software as its executing could take one of the several different execution paths that it goes through and based on the path that it takes it could end in one of the many different states that it is in. So, typically graph models that represent software artifacts we will have more than one final state. In this example that I have shown in this slide it so happens that both these graphs have exactly one final state, but that need not be the case in general.



So, what is the summary of this slide? So, certain vertices in graphs which occurs models of software artifacts could be marked as special initial vertices from where computation is suppose to begin. We will identify them by this incoming line, which is not connected to any vertex on the other end and some vertices are marked as special final vertices which are marked by this double circles as you can see in these two figures and they are supposed to represent vertices in which computations end. Another point to note is that there could be graphs in which both initial vertex and one of the initial/final vertices is also an initial vertex it is nothing that is specified which says the set of the initial vertices and final vertices should be disjoint there is no requirement like that right.

(Refer Slide Time: 11:06)



Graphs in Testing: Coverage Criteria

- Graphs are very popularly used structure for testing.
- Graphs can come from different software artifacts:
 - Control flow graphs
 - Data flow graphs
 - Call graphs
 - Designs modelled as finite state machines and statecharts.
 - Use case diagrams
 - Activity diagrams
- Most of these graphs will have **labels** associated with vertices and/or edges. Labels or annotations could be details about the software artifact that the graphs are modelling.
- Tests are intended to cover the graph in some way.

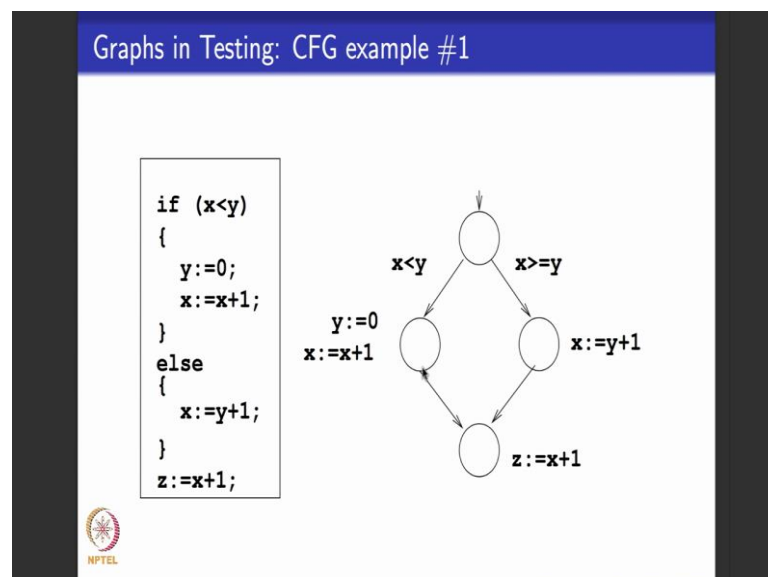
 

So, how are we going to use them as graphs? Why do we need look at graphs as far as software testing is concerned? Graphs, I believe, are next to logical predicates, may be a very popular structure used in testing right the several testing and static program analysis tools use graph models of software artifacts. So, where do these graphs come from? They could come from several different sources in software artifacts they could represent control flow graph of a particular program. Do not worry if you do not know these terms will introduce each of these terms as we move on in the course. They could represent what is called a data flow graph corresponding to a piece of code, they could represent what is called the call graph corresponding to a piece of code.

They could represent a software design element which is a modeled let say as an UML finite state machine or a UML state chart, they could represent a requirement which is given as a use case diagram or an activity diagram in the UML notation. All these are basically graph models that represent several different artifacts. Now you might ask a question, the kind of graphs that I define to you just a few minutes ago just had vertices edges and may be some vertices marked as initial vertices and some vertices marked as final vertices right. Obviously, the kind of graphs that I am talking about here through these different software artifacts and not going to be as simple as that. They we will typically have lot of extra annotations or labels as parameters right.

There could be labels associated with vertices there could be labels associated with edges and so on and so forth as and when needed we will look at a corresponding kind of graph, but no matter what the kind of graphs they are they will always have this underlying structure. We have vertices, some vertices marked as initial vertices, some vertices marked as final vertices and set of edges the edges could be directed or undirected. They will always have this structure and we will set of other things. And how are we going to use? Our goal is to be able to design test cases that we will cover this graph in some way or the other covering in the sense of coverage criteria that I defined to you in one of my earlier lectures.

(Refer Slide Time: 13:26)



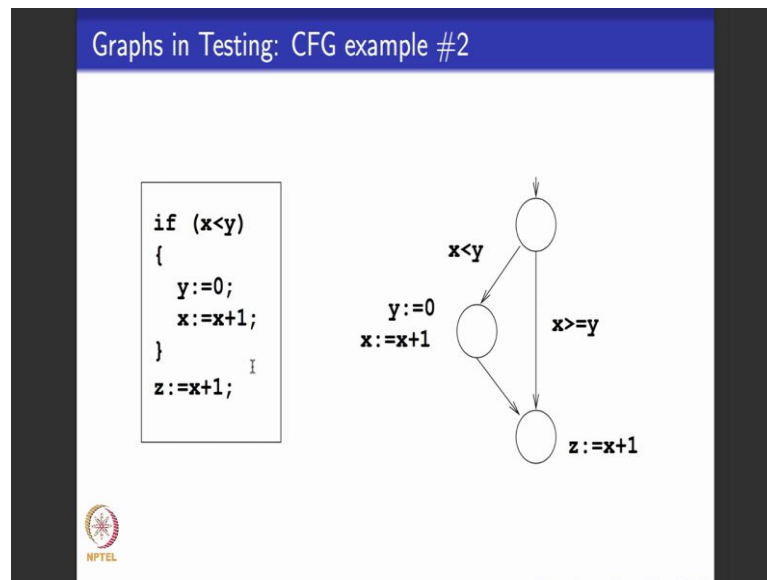
So, here is an example of how typical graph occurs as a model of a software artifact. Another thing that I would like to reiterate at this point in my course is that when we look at examples like this I will never show you a complete piece of code that is fully implemented; I will always show you a small fragment of code. Like if you see in this example I have just shown a small fragment of code this does not mean that this is the entire program. It can never be a full complete program and it does not make sense if it is a full complete program because you do not know what its inputs are, what its outputs are where are the outputs being produced its clearly not an implementation ready code right.

We will always look at fragments of code that is useful for us to understand a particular algorithm or a methodology for test case design. We will of course, see examples where we will see full pieces of code, but as I go through my lecture we will see a lot of code fragments. So, do not confuse them with the code that is ready for implementation, it will not be the case. So, here is a piece of code fragment which talks about an if statement. So, there is an if statement which says that if x is less than y which means if this predicate turns out to be true then you go ahead and execute these two statements what are these two statements one says assign 0 to y and the other says make x is equal to x plus 1 and with this predicate turns out to be false then you say you say x is y plus 1 and no matter what you do when you come out you make z as x plus 1.

So, here is a graph model corresponding to this particular code fragment. How does this graph model look like? So, corresponding to this first if statement there is an initial node in the graph which is marked here. This if statement basically tests for true or falsity of the predicate x less than y . So, if x is less than y , this code takes this branch. If x is not less than y which means x is greater than or equal to y then this code takes, but this branch. Suppose x is less than or equal to y then these two statements are to be executed. So, when it takes this branch, I model one collapsed control flow mode which basically represents the execution of two statements in order, in the order in which they occur. The two statements are as they come in the code--- assign 0 to y , assign x plus one to x .

Suppose x less than y was false, then the code takes the else branch and it comes here and it executes the statement x is equal to y plus 1. So, it does not matter whether it takes the then branch and next branch as per this example when it comes out of the if, it executes the statement. So, no matter whether it goes here or it here it always comes back and executes this statement. So, this is how we modeled controlled flow graph corresponding to a particular program statements. So, later I will show you for all other constructs, for loops and other things how those control flow graphs look like.

(Refer Slide Time: 16:33)



Here is another small example. Let say suppose you take the same if statement, but it in have the else clause right. So, there is nothing that specified in the code about what to do when this condition in this predicate x less than y is false. It does not matter, people can write code like that in that case what you do is if x is less than y from here this node which represents this if statement you do these two statements at this node which is assign 0 to y and assign x plus 1 to x and if this condition x less than y is false then you come directly and execute the statement z is equal to x plus 1.

So, if you see this kind of graph the graph that we saw in this slide or the graph that we saw in this slide they have vertices as we saw then they have a designated initial vertex. I have not marked any vertexes final vertex because I do not know whether the computation of this code fragment ends here as a part of the larger code or not, And, in addition to that if you see both vertices and edges have labels associated with them. These vertices, this vertex is labeled with two statements from the program this vertex is labeled with one statement from the program, these two edges are labeled with what are called guards or predicates that tell when this edge can be taken. So, like this typically all other models of graphs that we will look at, which is from this list, we will always look at some kind of extra annotations labels that comes with these kind of structures.

(Refer Slide Time: 18:03)

Paths in Graphs

- A **path** p is a sequence of vertices v_1, v_2, \dots, v_n such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq n-1$.
- **Length** of a path is the number of edges that occur in it. A single vertex path has length 0.
- **Sub-path** of a path is a sub-sequence of vertices that occur in the path.
- For e.g., for the undirected graph given in slide 3, u, w, x, u, v is a path. A sub-path of this path is w, x, u and the length of this sub-path is 2.

NPTEL

So, another important concepts that we need related to graphs as its used in testing is a notation of path in the graph. Think of a path as in the graph a sitting at some vertex or a node and just using the I just to walk through the graph. So, what is a path? Path is a sequence of vertices, just a finite sequence of vertices. I told you will consider finite graphs; finite graphs does not mean that they give rise to finite paths because a graph could have a cycle where in you can take it again. Cycle is a path that begins and ends at a same vertex and you could take it again and again to be able to get an infinite path, but we will look at finite paths for now. So, a path is a finite sequence of vertices let us say v_1, v_2, \dots, v_n , such that each pair of successive vertices in the path is connected by an edge.

So, if I go back here to the example graphs that we saw in the first slide right - here is a path and this graph it begins that u , from u I go to w , from w I go to x , from x I can go back to u and let say from u I go to v right. So, this is a path in the graph. So, what is the length of the path? When you talk about the length of the graph we count number of edges in the paths? So, the length of the path is a number of edges single vertex could be a paths and the length of such a path will be 0, right. So, what is a sub-path of a path? A sub-path of a path is just a subsequences of vertices that occur in the path. If we go back to that example graphs that we had in mind, I told you u, w, x, u, v is a path right.

(Refer Slide Time: 19:57)

Reachability in graphs

- A vertex v is **reachable** in a graph G if there is a path from one of the initial vertices of the graph to v .
- An edge $e = (u, v)$ is **reachable** in a graph G if there is a path from one of the initial vertices to the vertex u and then to v through the edge e .
- A sub-graph G' of a graph G is **reachable** if one of the vertices of G' is reachable from an initial node in G .

NPTEL

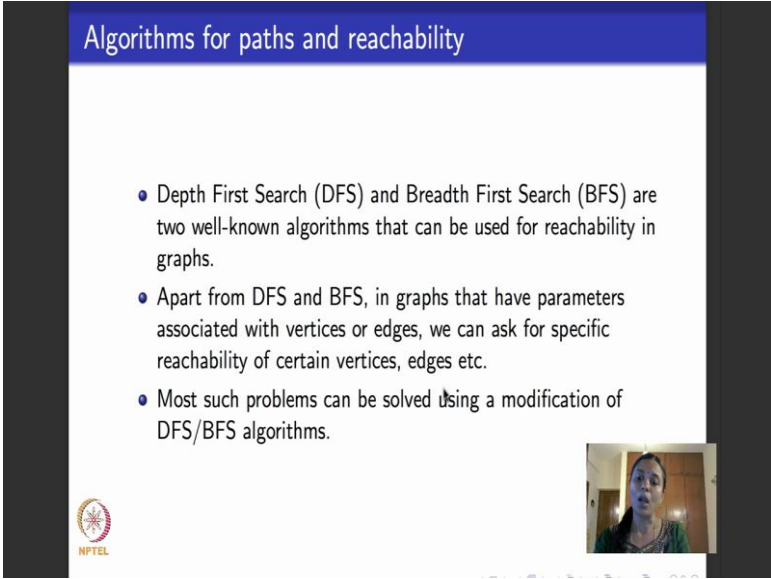
In this there is a sub path which is say $w \rightarrow x \rightarrow u \rightarrow v$. That is a sub path, $u \rightarrow w \rightarrow x$ is another sub path. So, it is just sub sequence in the sequence of vertices that you encounter in the path. So, why are we looking at paths? We will look at paths because we have going to be able to design test cases that we can use these paths to reach a particular statement or a particular node in the graph corresponding to that software artifact. So, we say a particular vertex v is reachable in the graph if there is a path from one of the initial vertices of the graph to v . For the sake of reachability, we will consider those paths that begin at initial vertices only because we want to be able to satisfy the RIPR criteria if you remember. RIPR - the first R is reachability. So, reachability means from the inputs from the initial state of the corresponding graph, I should be able to reach a particular vertex and moving on I should be able to propagate the output to a particular vertex right. Those final vertex to which the output is propagated and visible to the user would be one of the final vertices.

So, we say a particular vertex v is reachable in the graph if there is a path from one of the initial vertices to that vertex v in the graph. Reachability is not restricted to just vertices you could talk about reachability for an edge also. So, when is an edge reachable in a graph? We say a particular edge is reachable in a graph if there is a path from one of the initial vertices to the beginning vertex of that edge which is u and moving on, it actually uses that edge to reach v right. So, there is a path from one of the initial vertices to the edge, to the vertex u and then it uses the edge $u \rightarrow v$ to be able to reach the vertex v

then you say that the edge e which is given by the pair u comma v is reachable in the graph.

I hope you know the notion of a sub graph of a given graph. So, what is a sub graph of the given graph? It has a subset of the set of vertices and then it has a subset of the set of edges restricted to only those vertices that occur in the graph. So, you should go back and look at the same example that we had. So, here is a graph, I can think of just this triangular entity right which consist of three vertices u x w and these three edges as a sub graph of this entire graph. Or, you could just considered just this stand alone vertex v and another stand alone vertex y as a sub graph just containing two vertices. So, we can talk about reachability for sub graphs also. So, we say a sub graph G prime of a graph G is reachable if any of the vertices in that sub graph is reachable from the initial vertex of G .

(Refer Slide Time: 22:41)



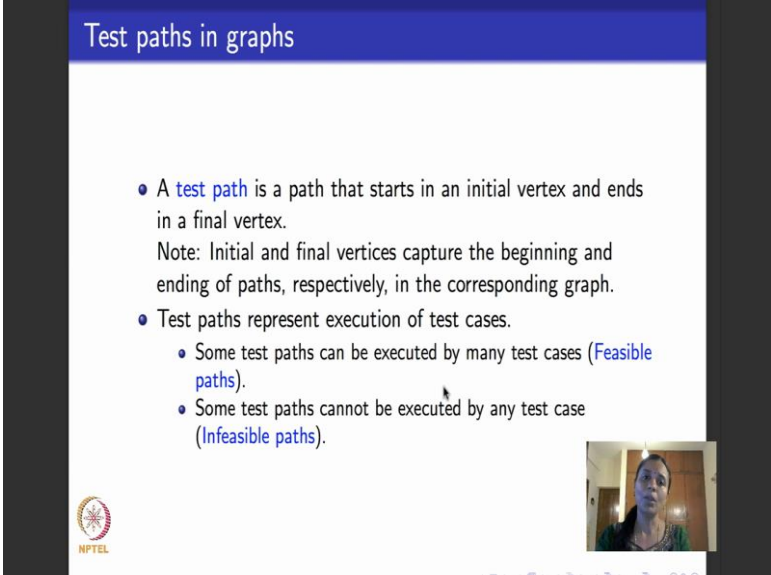
The slide is titled "Algorithms for paths and reachability" in a blue header. It contains three bullet points: "Depth First Search (DFS) and Breadth First Search (BFS) are two well-known algorithms that can be used for reachability in graphs.", "Apart from DFS and BFS, in graphs that have parameters associated with vertices or edges, we can ask for specific reachability of certain vertices, edges etc.", and "Most such problems can be solved using a modification of DFS/BFS algorithms." In the bottom right corner, there is a small video inset showing a woman speaking. The NPTEL logo is in the bottom left corner.

- Depth First Search (DFS) and Breadth First Search (BFS) are two well-known algorithms that can be used for reachability in graphs.
- Apart from DFS and BFS, in graphs that have parameters associated with vertices or edges, we can ask for specific reachability of certain vertices, edges etc.
- Most such problems can be solved using a modification of DFS/BFS algorithms.

So, are there algorithms that deal with computing paths and computing reachability of a particular vertex? Of course, all I am assuming all of you know basic graph algorithms in case you do not know please feel free to look up NPTEL courses the deal with design and analysis of algorithms and get to know about graph algorithms. Two basic algorithms that you have to be familiar with what are called breadth first search and depth first search. Most of the test case design algorithms that we will deal with in this course will involve depth first search or breadth first search along with some

manipulations and add on to these algorithms. I will not be able to cover these algorithms because I want to be able to focus on test case design using graphs.

(Refer Slide Time: 23:29)



The slide is titled "Test paths in graphs" in a blue header. It contains the following content:

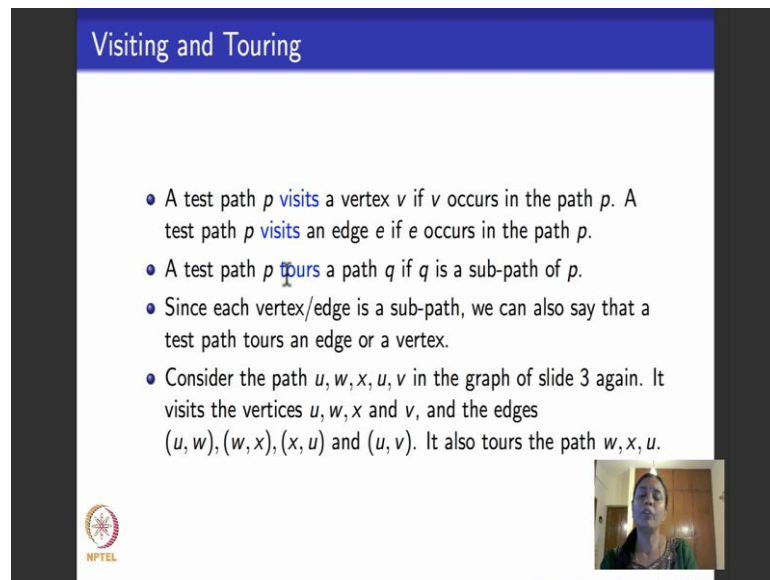
- A **test path** is a path that starts in an initial vertex and ends in a final vertex.
Note: Initial and final vertices capture the beginning and ending of paths, respectively, in the corresponding graph.
- Test paths represent execution of test cases.
 - Some test paths can be executed by many test cases (**Feasible paths**).
 - Some test paths cannot be executed by any test case (**Infeasible paths**).

In the bottom right corner of the slide, there is a small video inset showing a woman speaking. The NPTEL logo is visible in the bottom left corner of the slide area.

So, now instead of looking at arbitrary paths and graphs we will see what are test paths in graph. So, what is a test path a test path as I told you has to begin in an initial vertex to be able to ensure reachability and it has to end in one of the final vertices. So, a test path in a graph is any path that begins in an initial vertex and ends in a final vertex. So, if I go back to the same example, if I see here path of the form $u \rightarrow x \rightarrow w$ is a test path because it begins in an initial vertex u and ends in a final vertex w , whereas path of the form $u \rightarrow w \rightarrow x \rightarrow y$ is not a test path because even though it begins in initial vertex u , the vertex that it ends which is y , is not one of the final vertices. So, for us, test paths will always begin at initial vertex and end at a final vertex.

So, test paths will result in some test cases being executable. Test paths, if some test paths can be executed by test cases then those of called feasible test paths. There could be test paths for which I cannot execute any test case, like for example, there could be a test path which says you somehow reach a piece of dead code or unreachable code. So, I will not be able to write a test case that you can reach the dead code with. So, such test paths will be called as infeasible test paths. We will make these terminologies clear as we move on.

(Refer Slide Time: 25:04)



The slide is titled "Visiting and Touring" in a blue header. It contains a bulleted list of definitions and a video inset in the bottom right corner showing a person speaking. The NPTEL logo is in the bottom left corner of the slide area.

- A test path p **visits** a vertex v if v occurs in the path p . A test path p **visits** an edge e if e occurs in the path p .
- A test path p **tours** a path q if q is a sub-path of p .
- Since each vertex/edge is a sub-path, we can also say that a test path tours an edge or a vertex.
- Consider the path u, w, x, u, v in the graph of slide 3 again. It visits the vertices u, w, x and v , and the edges $(u, w), (w, x), (x, u)$ and (u, v) . It also tours the path w, x, u .

So, few other terminologies we need to be able start looking at algorithms for test case design. Those are the notions of visiting and touring. So, we say test path p visits a vertex v , v occurs along the path p right. Similarly, a test path p visits an edge e if e occurs along the path p right.

A test path p tours path q if q happens to be a sub path of p . We will go back to the same graphs that we looked at, so here is a test path right $u \rightarrow x \rightarrow w$, right. So, this test path visits three vertices $u \rightarrow x$ and w and it visits two edges the edge $u \rightarrow x$ and the edge $x \rightarrow w$ and it tours a sub path $w \rightarrow x$. You could consider another test path which looks like this it could be $u \rightarrow w \rightarrow x \rightarrow u \rightarrow w$ right. So, this test path visits three vertices $u \rightarrow x$ and w it happens to visit them again and again, but basically it visits only three vertices and it tours the sub path $u \rightarrow w \rightarrow x$. So, I hope visiting and touring a clear value we will look at what are tests and test paths.

(Refer Slide Time: 26:24)

Tests and test paths

- When a test case t executes a path, we call it the **test path** executed by t , denoted by $path(t)$.
- Similarly, the set of test paths executed by a set of test cases T is denoted by $path(T)$.

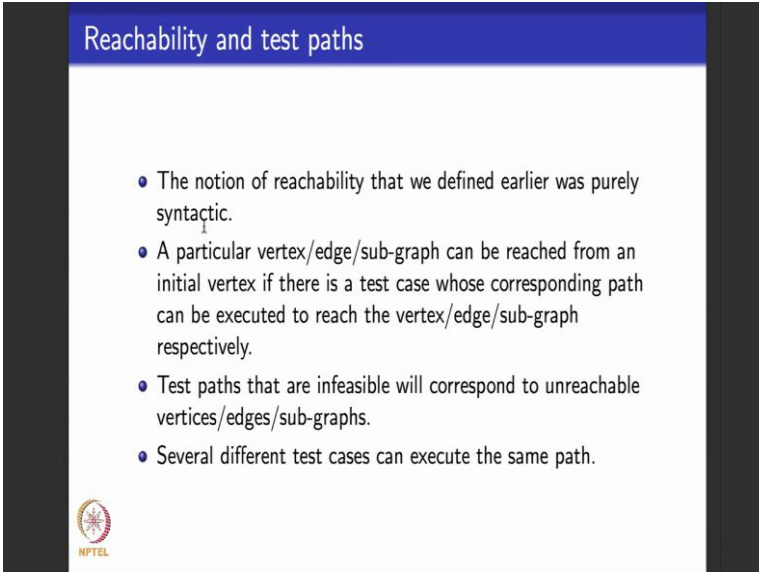
Test case input: $(a=0, b=1)$ Test path: u, v, x, x
Test case input: $(a=1, b=1)$ Test path: u, w, x
Test case input: $(a=3, b=1)$ Test path: u, x

Let us take a small example look at this graph this graph is got four vertices, u v w and x and here is an initial vertex u . It models control flow corresponding to a simple switch statement switch case statement, the switch case statement is at the node u . There are three switch cases, if a is less than b you go to v execute may be some statements, if a is equal to b then you go to w and do something else if a is greater than b then you go to x and then you do something else and it so happens that computation terminates at x . So, what suppose I have a test case input a as 0 and b as 1 then which is the condition that it satisfies? It satisfies a less than b right.

So, which is the test path that this test case executes? This test case executes the path u v and remember test path is one that has to end in a final state right. v is not one of the final states. Which is a final state in this graph? It happens to be the state x right. So, I go from u to v because my test case satisfies the predicate a less than b and I can freely go from v to w and then from w to x because these two edges and this graph do not have any guards or conditions labeling them. So, if my test case input is a 0 and b 1 then I say that the test path that it takes is u to v because it satisfies a less than b and then v to w and w to x . This is clear? So, similarly if my test input is less say a is 3 and b is 1 then in this switch case statement the predicate that it satisfies is this: a greater than b and then the test path that it takes is just the single edge u x it just. So, happens that this path containing just this one edge already is a test path because it begins at the initial vertex u and ends at the final vertex x .

So, similarly when I have set of test cases I can talk about a set of test paths. For each test case in the set of test cases you consider the test paths that they execute and take the union of all the test paths to be able to get the set of test paths corresponding to a set of test cases. Please remember that one test case can execute many test paths. This example does not show that, but one test case can execute. Like for example, if a club these two conditions right and say a is less than or equal to b then I can write several test cases write that will execute this path.

(Refer Slide Time: 29:07)



Reachability and test paths

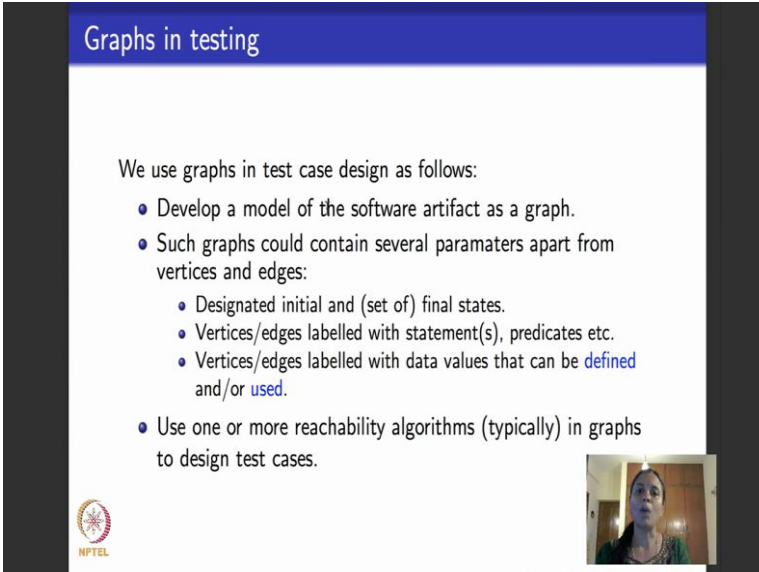
- The notion of reachability that we defined earlier was purely syntactic.
- A particular vertex/edge/sub-graph can be reached from an initial vertex if there is a test case whose corresponding path can be executed to reach the vertex/edge/sub-graph respectively.
- Test paths that are infeasible will correspond to unreachable vertices/edges/sub-graphs.
- Several different test cases can execute the same path.

NPTEL

Now, we will go back and see reachability in the context of test paths. When we define reachability for graphs we defined it in a purely syntactic nature you say a particular vertex b is reachable iff there is a path from the initial state to one of the vertices right. As you see in this example it need to be purely syntactic because there could be conditions associated with edges, there could be something associated with vertices. As you traverse along a path your implicitly allow to traverse only if that condition a guard is met right. So, in the real application of graphs we would be looking at reachability in the presence of these additional annotations and conditions. So, we will see what reachability means in the presence of these conditions. And it is to be noted that if I have a particular test path that is infeasible, like I told you right a test path that insist that you reach a piece of dead code than it corresponds to vertices or nodes or edges or graphs that are not reachable from the main graph.

Like for example, it might so happen that the control flow graph of a particular program has two disjoint components, in which case suppose I insist that all the initial vertices are in one component, I insist that you reach a vertex from the initial vertex to another vertex in the second component. Because these two are two disjoint components and may not be able to reach that vertex at all and I say that it represents an infeasible test requirement. So, we will see several examples of where infeasible test requirements could occur when we look at graph models. So, how are we going to use graphs for test case design?

(Refer Slide Time: 30:44)



Graphs in testing

We use graphs in test case design as follows:

- Develop a model of the software artifact as a graph.
- Such graphs could contain several parameters apart from vertices and edges:
 - Designated initial and (set of) final states.
 - Vertices/edges labelled with statement(s), predicates etc.
 - Vertices/edges labelled with data values that can be **defined** and/or **used**.
- Use one or more reachability algorithms (typically) in graphs to design test cases.

NPTEL

(A small video inset in the bottom right corner shows a woman speaking.)

We will develop a model of a software artifact as a graph, I already showed you to examples of how to take a code snippet and write a control flow graph corresponding to that. It is to be noted as we saw in that example that these graphs apart from vertices and edges could contain several different annotations labels. So, here are some examples some of the vertices could be initial and final vertices it could have a labels of statements predicates associated with its vertices and edges as we saw in those two examples. In addition to that, it could have data values data values for variables that are defined at particular statement, data values for variables that I used at a particular statement. Such graphs are called data flow graphs I have not shown you an example in this slide, but in later lectures we will see what data flow graphs look like.

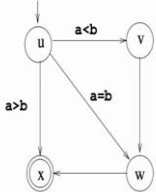
But what is to be remembered is that they will typically, graphs that model software artifacts will always have some kind of labels associated with their vertices or edges

right. So, we want to be able to use one or more reachability algorithms to be able to design test cases for these kind of graphs.

(Refer Slide Time: 31:57)

Graph coverage criteria

- **Test requirement** describes properties of test paths.
- **Test Criterion** are rules that define test requirements.
- **Satisfaction:** Given a set TR of test requirements for a criterion C , a set of tests T satisfies C on a graph iff for every test requirement in $t \in TR$, there is a test path in $\text{path}(T)$ that meets the test requirement t .
- For example, the set of test cases below in the graph satisfy **branch coverage** at the node u in the graph.



Test case input: (a=0, b=1) Test path: u, v, x, x

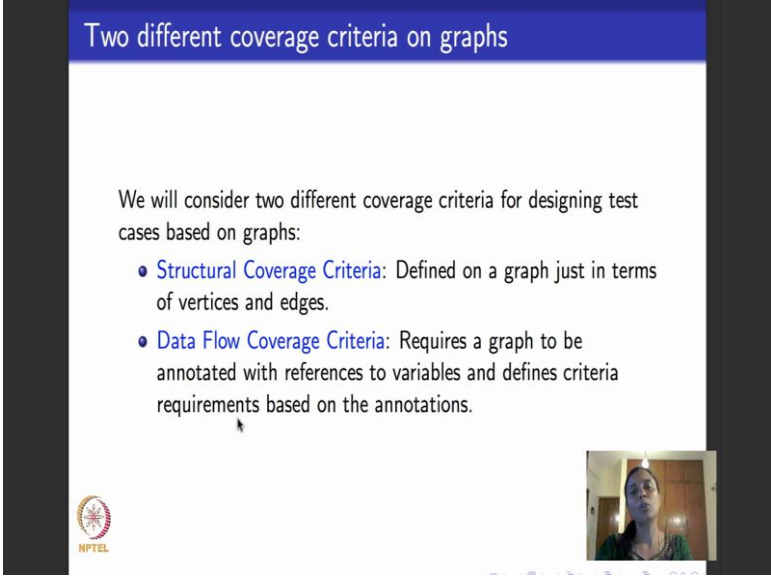
Test case input: (a=1, b=1) Test path: u, w, x

Test case input: (a=3, b=1) Test path: u, x

So, we will read a find what test requirement what test criteria are now specific to graphs. So, what is a test requirement? It just describes a property of a test path, test path as it corresponds to in a graph. What is a test criteria? Test criterion is set of rules that define the test requirement. Then, for example, if you take this graphs that we looked at earlier the test criterion that it describes is to say do branch coverage on the vertex u means the degree of vertex is the out degree of the vertex u is 3, there are three branches going out of u . Write test cases to cover all three branches that is what we did here we do not test cases to cover all three branches.

So, what is satisfaction? We see a particular set of test requirement satisfies a coverage criteria c if the test cases that are right for that test requirement satisfy all the test paths that need the requirement. Like for example, if my coverage criteria says branch coverage at mode u then, these three test cases right completely achieve branch coverage for this particular mode u .

(Refer Slide Time: 33:10)



Two different coverage criteria on graphs

We will consider two different coverage criteria for designing test cases based on graphs:

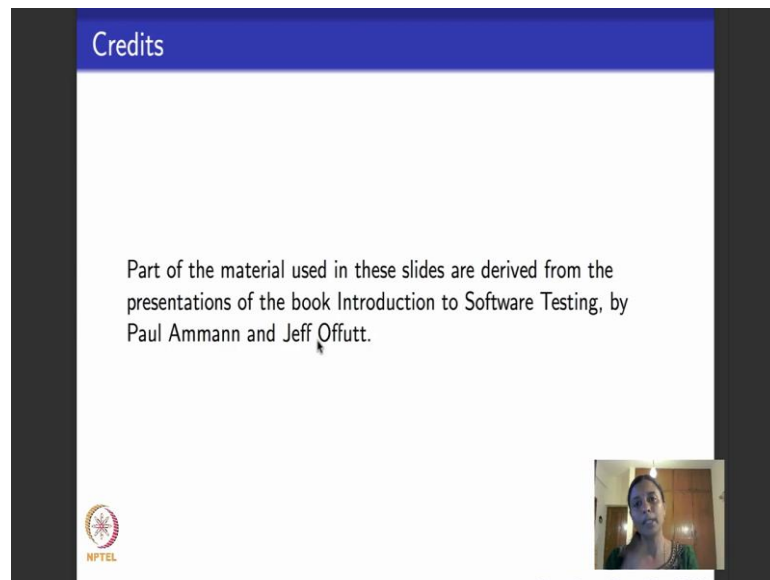
- **Structural Coverage Criteria:** Defined on a graph just in terms of vertices and edges.
- **Data Flow Coverage Criteria:** Requires a graph to be annotated with references to variables and defines criteria requirements based on the annotations.

NPTEL

So, moving on we will look at two kinds of coverage criteria related to graphs. The first coverage criteria that we will be looking at is what is called structural coverage criteria where we will define coverage criteria purely based on the vertices and edges in the graph.

They could be annotated with statements and so on, but we would not really use the what the statements are or the other annotations to be able to define the coverage criteria. We will purely define in terms of just vertices and edges. The next kind of coverage criteria would be data flow coverage criteria where we again look at graphs that are annotated with variables and so on and we will define coverage criteria based on the annotations, based on the variables and the values that they define.

(Refer Slide Time: 34:02)



So, in the next module I will begin with structural coverage criteria and walk you through algorithms and test case design for achieving various kinds of structural coverage criteria in graphs.

Thank you.

Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 06
Structural Graph Coverage Criteria

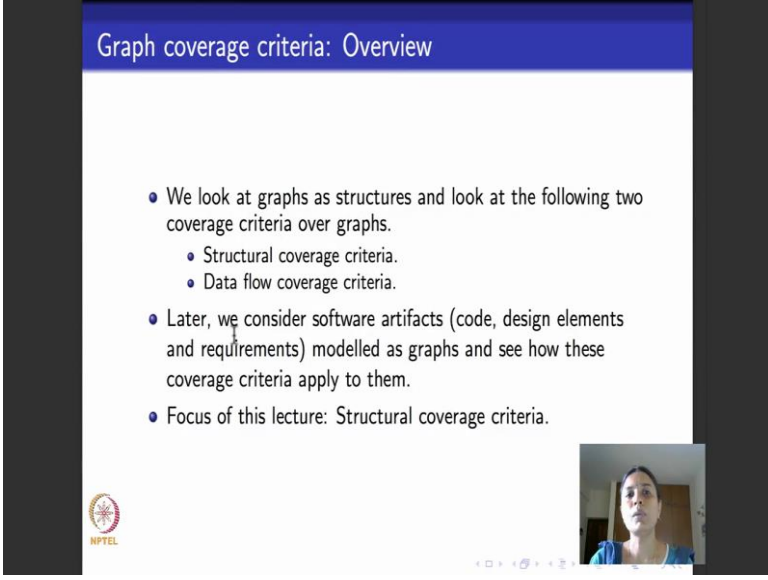
Hello everyone. Now we are in the second week. What we looking at this week is basically you look at graphs and to look at various algorithms that we can use for test case design using graphs. In the last module I gave you a brief background of graphs as it was necessary for this course. So, to recap what we saw in the last module: we saw what graphs where and in the basic concepts like degree of vertex, what is the notion of paths, and what are trips, what is visiting a node and few are the details. So, the plan is to be able to use graphs to design test cases. So, we take software artifacts like code, requirement, design, model them as graphs and see how to design test cases, how to define coverage criteria and then to design test cases using graphs.

So, in the first two modules that we will see as a part of graph based testing what I will do is we look at coverage criteria purely based on graphs. I will not really show you too many examples of how software artifacts are modeled as graphs. Instead we directly deal with graphs as data structure it is and define coverage criteria. In today's lecture we will define coverage criteria that are based on the structure of the graphs. That are based on nodes, vertices, edges, paths and so on.

The next lecture I will look at some algorithms and deal with coverage criteria that are again based on the structure of the graph. Moving on what we will do is when annotate the graph the vertices and the edges of the graph with statements and other entities. And we will define coverage criteria based on data that deals it with in the graphs. After we do these three kinds of coverage criteria, look at how they are related to each other, and how they can be use to design test cases based on graphs we will consider a later module where we will take various kinds of software artifacts; we will begin with code, then to go on to design, then move on to requirements.

Model each of them as graphs and then see how the coverage criteria that we will be learning throughout these modules can be used to actually design test cases for covering the software artifacts.

(Refer Slide Time: 02:29)



Graph coverage criteria: Overview

- We look at graphs as structures and look at the following two coverage criteria over graphs.
 - Structural coverage criteria.
 - Data flow coverage criteria.
- Later, we consider software artifacts (code, design elements and requirements) modelled as graphs and see how these coverage criteria apply to them.
- Focus of this lecture: Structural coverage criteria.

NPTEL

A small video inset in the bottom right corner shows a woman with dark hair, wearing a blue top, speaking.

So, what are we be doing at. So, first I will today's module I will deal with structural coverage criteria, in the next module also we look at structural coverage criteria but we will focus on algorithms, and then we will look at what is called data flow coverage criteria which consider graphs with data, variables and they values and then define coverage based on them. As I told you post this, we will take it has several software artifacts one at a time module them as a graphs and see how to use these coverage criteria to design test cases. So, we begin with coverage criteria in this course.

(Refer Slide Time: 03:03)



Structural coverage criteria over graphs

- Node/vertex coverage.
- Edge coverage.
- Edge pair coverage.
- Path coverage
 - Complete path coverage.
 - Prime path coverage.
 - Complete round trip coverage.
 - Simple round trip coverage.

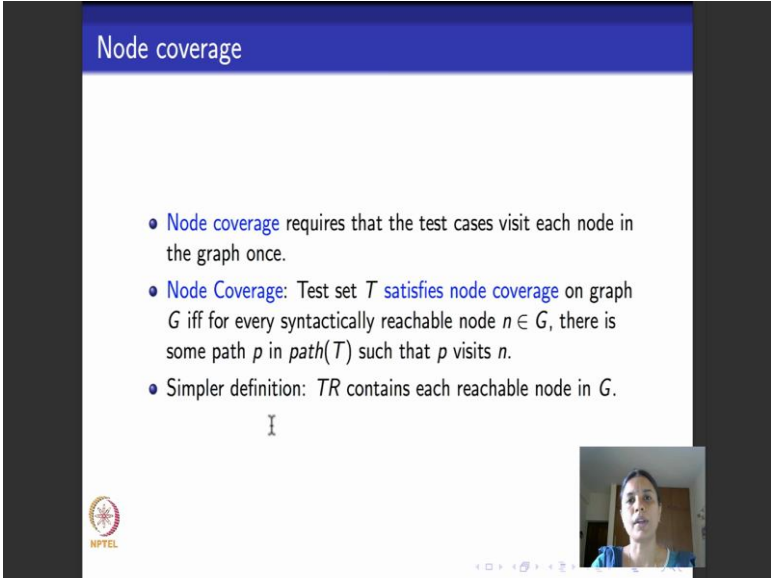
NPTEL

A small video inset in the bottom right corner shows the same woman from the previous slide, continuing her presentation.

So, what are the various coverage criteria that we are going to see? Here is the listing. So, we will begin with what is called node coverage or vertex coverage. We will define a test requirement which says you have to cover/visit every node. And then we will define test coverage requirements for edge coverage where we say we visit every edge. Then we look at edge pair coverage which insists on visiting every consecutive pairs of edges which are paths of lengths two. Then we look at path coverage in the graph. In path coverage these are the various things that we look at. We look at complete path coverage which may not be feasible all the time.

And we will move on and look at very popular coverage criteria called prime path coverage. And then to make prime path coverage feasible we look at these round trip coverage criteria. So, we will see each of these one at a time.

(Refer Slide Time: 03:59)



Node coverage

- **Node coverage** requires that the test cases visit each node in the graph once.
- **Node Coverage:** Test set T satisfies node coverage on graph G iff for every syntactically reachable node $n \in G$, there is some path p in $path(T)$ such that p visits n .
- **Simpler definition:** TR contains each reachable node in G .

So, we begin with node coverage, what is node coverage? It simply insists that you have a set of test cases that visit every node in your graph. So, it could be the case that the graph corresponding to a particular software artifact can be connected or disconnected. If it is disconnected then the graph has several disjoint components. What do we mean by disjoint components? There are no edges between vertices of these components.

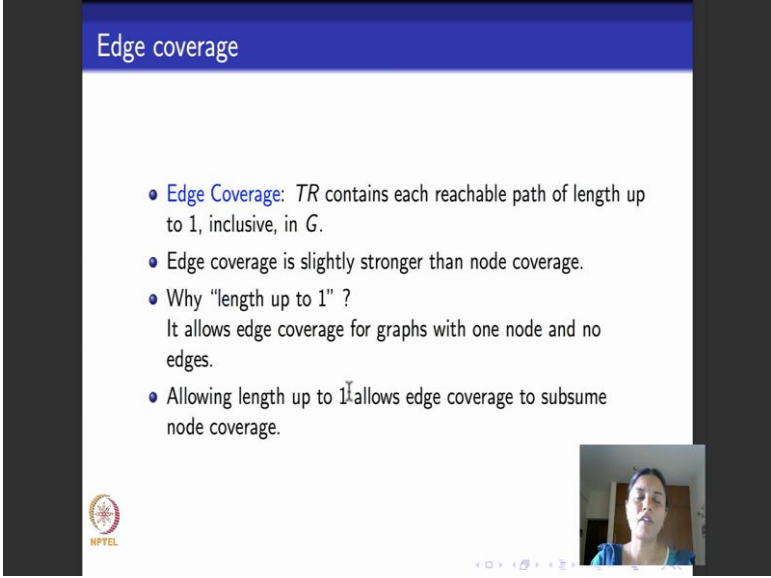
So, in which case if I see node coverage needs to visit every node, you might ask how do I go and visit other nodes then the graphs. So, what we insist is the node coverage visits every reachable node; every reachable node means every node that can be reached from

the designated initial node. So, what we mean is that per component per reachable component that is connected, you visit every node.

So, what is node coverage? Node coverage says, the test requirement for node coverage abbreviated as TR, says you basically right a set of test cases that will visit each node in the graph. Now how will you write a set of test cases that will visit each node in the graph? What will a test cases look like? How can you go about visiting nodes in a graphs. You have to start from a initial state and you have to walk along paths in the graphs. As you walk along paths you visit various different nodes.

So, set of test cases that will meet the test requirement on node coverage would be a set of test paths that begin at an initial state and visit every reachable node in the graph. I will show you an example after a couple of slides.

(Refer Slide Time: 05:40)



Edge coverage

- **Edge Coverage:** TR contains each reachable path of length up to 1, inclusive, in G .
- Edge coverage is slightly stronger than node coverage.
- Why "length up to 1" ?
It allows edge coverage for graphs with one node and no edges.
- Allowing length up to 1 allows edge coverage to subsume node coverage.

NPTEL

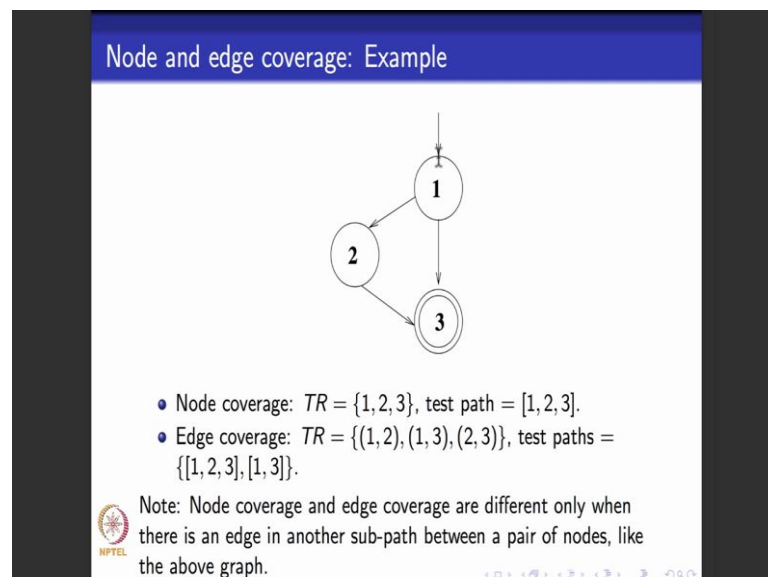
And then the next coverage criteria that we will be looking at; is what is called edge coverage. So, light node coverage edge coverage basically insists that your visit every edge in the graph. So, your test requirement says you visit each requirement path of length up to 1. So, you might ask when I say visit every edge why am I writing it as visit each reachable path of length up to 1? Now look at an edge in a graph what is an edge look like an edge can be thought of as a path of length 1, because which is connects two vertices. But instead of saying visit every path of length 1 which basically means visit every edge, we are slightly modifying the conditions to say visit every path of length up

to 1, which means you visit paths of length is 0 and paths of length 1. What do paths of length 0 correspond too? Paths of length 0 correspond to paths the just contains single vertex.

We want to be able to say that edge coverage subsumes node coverage in the sense that if I have a set of test paths that satisfy edge coverage then those set of test paths will also satisfy node coverage. Because we want to be able to say so, we modify edge coverage criteria definition to include that it visits every path of length up to 1.

So, it visits every single vertex and it visits every single edge. So, by definition edge coverage will subsume node coverage.

(Refer Slide Time: 07:17)



Here is an example to illustrate how edge and node coverage work. So, here is a small three vertex graph: vertex one is the initial vertex. As you can see, its mark to design coming arrow. Vertex three is the final vertex it is marked with its double circle and it is a small graph. If you remember in the previous example we had a new statement and I showed you how to model the control flow graph corresponding to that if statement remember, which did not have a else part just had the then part. This was the graph that correspond to the control flow graph of a that if statement.

I of course, remove the labels in the edges all that stuff, because we are looking at purely structural coverage criteria which defines coverage criteria based on the basic structure

of the graph; does not really look at labels of edges of vertices. So, in this graph what would be node and edge coverage. Node coverage: the test requirement for node coverage says that there are three nodes so you please visit all three reachable nodes, all three of them are reachable here. So, how many paths can visit all three reachable nodes? So, I could take this path I start from 1 and node 2 and then to 3. In this path I have visited all the three nodes 1, 2 and 3. There is another possible path in the graph I could start from 1 and then I go to 3. Suppose I take this as the test path corresponding to the test requirement then I have only partially achieved node coverage, I have visited the nodes 1 and 3 I have not visited the nodes 2.

There is no harm in choosing that, suppose you choose that then you have to add this path also 1, 2 and 3. But instead, I could directly choose this path 1, 2 and 3 as my test path. And in that case I would directly met the test requirement of node coverage which is one path.

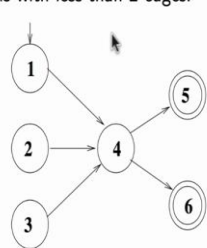
Now, let us look at edge coverage how many edges are there in the graph? There are three edges in the graph: one edge from 1 to 2, one edge from 1 to 3 and another edge from 2 to 3. What is edge coverage mean? The test requirement or TR, for edge coverage says you visit all the three edges. The edge 1, 2 the edge 1, 3 and the edge 2, 3. Here if you see suppose I take this path 1, 2 and 3. How many edges do I visit in this path? I visit 1, 2 and 2, 3. So, I visit two edges so that is this path here. But I have to be able to visit the edge 1, 3 so I have to take this path.

So, to meet the test requirement for edge coverage and this graph I need two test paths; the test path 1, 2, 3 and then the test path 1, 3. This is another small observation. Suppose the graph is a sort of a straight line right, there are no branching and it just to corresponds to some core the corresponds to sequence statements. It just like one linear order or a straight line or a chain. In that case node and edge coverage are same, because when I visit every edge there is only one path in the graph I visited the path and in the process I visited every node also. So, node and edge coverage become different in terms of the test paths that satisfy it only if there is a branching in the graph.

(Refer Slide Time: 10:24)

Covering Multiple Edges: Edge-Pair Coverage

- **Edge-Pair Coverage (EPC):** TR contains each reachable path of length up to 2, inclusive, in G .
- Paths of length up to 2 correspond to pairs of edges.
- Again, the phrase "length up to 2" ensures edge-pair coverage holds for graphs with less than 2 edges.



- $TR = \{[1, 4, 5], [2, 4, 5], [3, 4, 5], [1, 4, 6], [2, 4, 6], [3, 4, 6]\}$.
- Test paths are the same as above.

So, we move on. What is the next coverage criteria that were there in my list? Go back to that slide. So, we did node coverage which basically says visit every node. So, you write test path that visit every node edge coverage says visit every edge once, so you write test paths that visit every edge. Now will move on to edge pair coverage and then look at path coverage.

Before we move on to edge pair coverage I would like you to spend a minute thinking about what would node coverage, how would node coverage and edge coverage be useful. Assuming that such a graph is a control flow graph corresponding to a program what can you think of node coverage and edge coverage put together they basically mean you execute every statement in the program.

Because they insist that you visit every node and then they insist that you visit every edge. If you write a set off test path that satisfy this then you are writing a test path that basically execute every statement in the program. So, you are looking for a set of test cases that will exercise for test every statement in the program. Node coverage and edge coverage might be quite difficult to achieve, because for large pieces of program where the control flow graph is fairly large, writing a set of test cases that will achieve execution of every statement, every node or every edge, can be a large set of test cases and sometimes infeasible also.

So, we look at other state of test cases. The next set of structural coverage criteria that I would like to talk about is what is called edge pair coverage. So, as a name says, your test requirement here is to actually consider pairs of edges; not pairs of edges that are far away from each other but pairs of edges that are consecutive to each other. In other words, you consider paths of length up to 2. Paths of length up to 2 include paths of lengths 0 which include nodes, paths of length 1 which are edges and paths of length 2 which are actually sets of edges that occur one after the other. So, if you take this example graph, how many paths of lengths two are there? All these six paths are paths of lengths 2. So, I start from 1 which is an initial vertex and 5 and 6 are final vertices. So, I go from 1 to 4, 4 to 5 that is a path of length 2 pair of edges then I go from 1 to 4, 4 to 6 another pair of edges that are consecutive; 2, 4, 5, another pair of edges 2, 4, 6; 3, 4, 5; 3, 4, 6. So, my test requirement says- you visit all these paths of length two exactly once. So, here they have written length up to 2, but I have listed here paths of lengths 2. I implicitly assume that it includes paths of length 1 and paths of length 0.

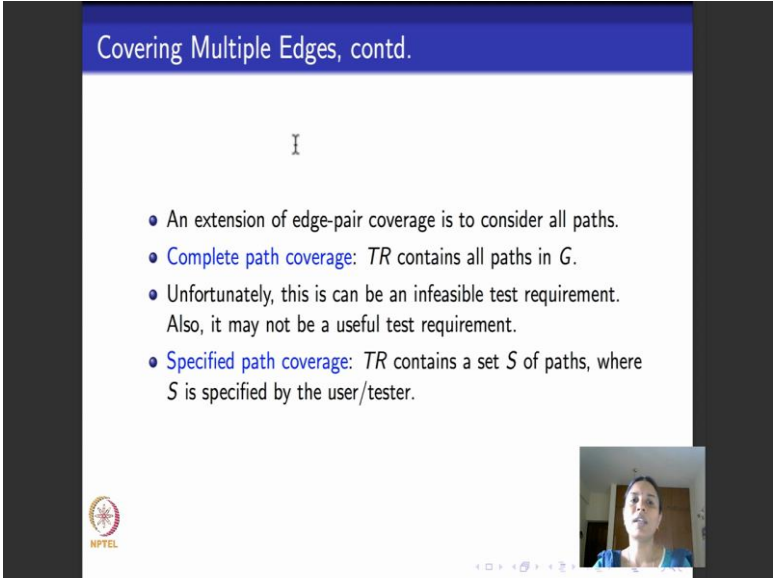
So, how many test requirements can I write? If you look at this structure of the graph it is so happens that I need one path for meeting each of these Trs, I cannot do better. So, test paths or basically all these six things, I cannot write anything shorter. Now why do I need paths of lengths up to 2, the same reason we said when we do edge coverage even though edge is a path of length 1 we changed it as path of length up to 1, because we wanted edge coverage to subsume node coverage. So, for the same reason we want edge pair coverage to subsume node coverage and edge coverage. So we insist that the TR contains all paths of length up to 2.

Suppose you do not you are not very particular about this requirement there is no harm in saying that edge pair coverage means, you visit all paths of length exactly 2. So now, before we move on what you think edge pair coverage would be useful for? One good use of edge pair coverage it is to cover all branches in the program. So, if you think of this as the control flow graph representing some piece of code, let's not worry about what that piece of code is, but let us assume that this is the CFG for some code.

What could node four be? Node four could be corresponding to an if statement right, which says it has two branches: if then is true may be you go to 5 if else true may be you go to 6 right due to something here you do something here. So, when I do edge pair coverage what I cover is from 1, 2 and 3, what are the various ways and which I can visit

4, and from 4, what are the various ways and which I can cover the two branches that go out of 4? So, edge pair coverage is useful. Suppose I think about it what did I do I did paths length 0 which was vertices no coverage, then I did paths of length 1 which were edges edge coverage. Now I say paths of length up to 2, I did edge pair coverage you can move on right you can say paths of the length 3, paths of length 4, paths of length 5. Then if we go on like this, what we have is what is called complete path coverage.

(Refer Slide Time: 15:17)



The slide is titled "Covering Multiple Edges, contd." and contains the following text:

- An extension of edge-pair coverage is to consider all paths.
- **Complete path coverage:** TR contains all paths in G .
- Unfortunately, this can be an infeasible test requirement. Also, it may not be a useful test requirement.
- **Specified path coverage:** TR contains a set S of paths, where S is specified by the user/tester.

The slide also features the NPTEL logo in the bottom left corner and a small video inset in the bottom right corner showing a woman presenting.

You have a test requirement TR which says that you cover all paths in the graph. Now if you think about it what would be the use of this? Will it be useful? What would all paths mean? If I say cover all paths on the graph let us assume a case where the graph has a loop. In the two examples that we saw in the previous two slides, the graph did not have a loop, but let us assume the graph has a loop. How many paths do you think will be there in the graph? They will be infinite number of paths in the graph, because I visit the loop once I get one path, and I visit the loop once again I will get another path, I visit a loop once again I get another path. So, I can go round the loop again and again and again and get more and more paths, longer and longer paths.

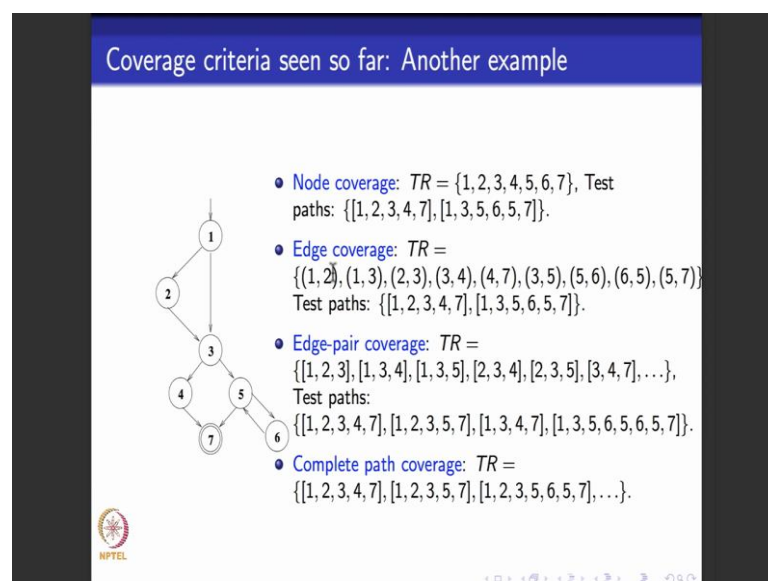
So, there will be infinite number of paths. And given these infinite numbers of paths how do I achieve complete path coverage? I will have to be able to go on writing test paths. So, for this reason complete path coverage is believed to be an infeasible test requirement. Infeasible in the sense that I will never be able to stop testing- if I want to achieve

complete path coverage intuitively. Also if you think about it not only is it infeasible, what is a use of it? It may not be very useful for us. What is the point you going on repeatedly executing a loop once, twice, thrice, four times and so on? It is not interesting, infeasible and not very useful.

So, what we say is that we will list complete path coverage as a test requirement, but it is not practically usable. What is practically usable is what is specify path coverage. What does specified path coverage say? It says that the use as a test engineer will give you the set of paths to cover. The TR will be a set of paths that is specified by a test engineer or a user which says you cover these paths. So, paths could be some special things, we do not know what they are, but the test is basically gives a set of paths and says you this is your test requirement. Please write a set of test cases that will cover this path.

So, we see specify path coverage, your test requirements contained a set of paths where that is set is specified by the user.

(Refer Slide Time: 17:44)



So, will go and now move on and see what is one specific, useful specified path coverage. Before that I will give you another example just to recap all the coverage criteria that we have seen so far. So, what are the folk structural coverage criteria that we saw so far? We saw node coverage, edge coverage, edge pair coverage and complete path coverage. So, let us take a small graph. Here is a graph on the left hand side, it has about 7 nodes, the node 1 is the initial node, and there is one final node which is node 7.

If you see this could constitute a reasonably interesting control flow graph. So, if you try to map it to assume that it is a control flow graph corresponding to some code there is branching at statement one. So, maybe there was an if statement here. There is another branching at statement 3. There is a another branching statement 5. One branching ends in the final state 7, one branching goes into a loop between 5 and 6. So, maybe there was a while statement here, this means skipping the loop and this means executing the loop.

So, what will node coverage on such a graph be? Node coverage on such a graph--- the test requirements says there are 7 nodes, please visit all 7 nodes. How many test path will visit all the 7 nodes? There could be one test path like this I do 1, 2, 3 4 7. This is the path, this test path from the initial node 1 to the final node 7.

So, in this test path I have visited how many nodes? I have visited 5 of my 7 nodes I visited nodes 1, 2, 3 4 and 7. What are the node 7 left behind; 5 and 6. So, I have to write another test path which includes that. So, again because it is a test path I have to begin from an initial state and end in a final state. So, I begin at node 1, then I go to 3, I do not want to go to 4 because I have already considered that in my test path; from 3 I go to 5; and then I do not want to go to 7 because if I do that then I will miss visiting node 6, so from 5 I visit node 6. And remember it is a test path so I have to be able to end in a final state. So, I come back to 5 and then I do 7, so that is the test path.

So, just to summarize node coverage test requirements says you visit all the 7 nodes which is the set. And what are the test paths that will visit all the 7 nodes there are two test paths one which goes like this 1, 2, 3, 4, 7 which leads of nodes 5 and 6. So, I put another test path which says 1, 3, 5, 6, 5, 7. So, these two test paths put together satisfy this test requirement of node coverage. Similarly on this graph how do I do edge coverage? How many edges are there? So many edges are there right; (1, 2); (1, 3); (2, 3); (3, 4); (4, 7) and so on.

So, test requirement says you visit all this edges which is the set of all edges in the graph. What are the two test paths a very similar to node coverage, because I told you right edge coverage also meant to subsume node coverage. So, if I do this test path 1, 2, 3 4 7 I visited all the edges that occur on this set. Now, I write another test path that lead covers a missing edges which is this (1, 3); (3, 5); (5, 6); (6, 5) and so on. So, between these I have done edge coverage the next test requirement is edge pair coverage. Edge pair

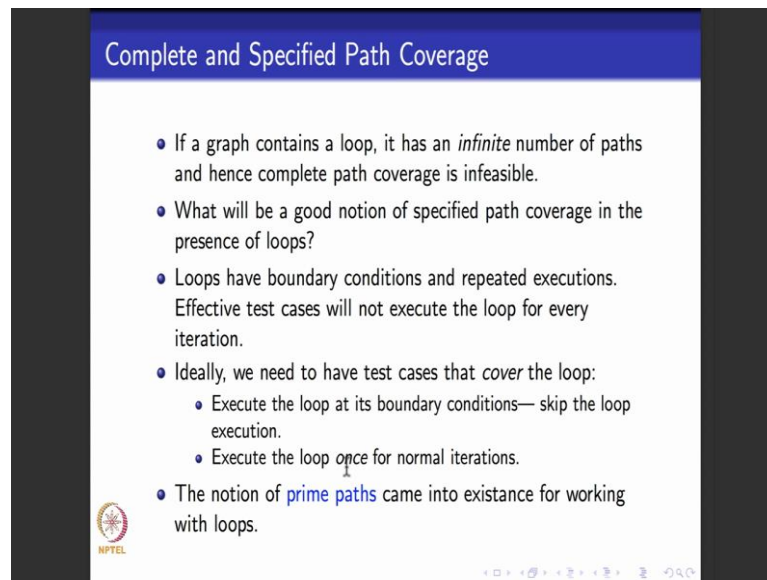
coverage lists all paths of length 2 which is 1, 2, 3; 1, 3, 4; 1, 3, 5 and so on that put this dot dot dot because I ran out of space to list all the edge pairs, but you can finish this its only the finite sets slightly larger than edge set that is my TR.

So, please write path then visit all this path of lengths 2 that would means it will visit all these consecutive pairs of edges. So, test paths for these would be if you see 1, 2, 3, 4, 7 it covers edge pairs that occurs a longest path and then I do 1, 2, 3, 4. I forgot this pair so 2 to a 3, 3, 5, so I do a 3, 5 7 which covers all test paths along these lines and then I do 1, 3, 4, 7 which includes this edge pair. Now I have to include 1, 3, 5, 6 5, 7 which completes all the edge pairs that I had.

So, with this four test paths I can achieve the test requirement for edge pair coverage. Now complete path coverage for this graph- please remember this graph has a loop here from 5 to 6. So, complete path coverage for this graph test requirement will not stop, it will go on listing paths, because I have this path it skips the loop, I have this path which also skips the loop for then I have take this path which enters the loop. Once I enter the loop I have a path that looks like this which visits the loop once. Then I can do the same thing again 1, 3, 5, 6, 5, 6, 5, 6, 7 and then I can do 1, 3, 5, 6, 5, 6, 5, 6, 5, 6, 7 then I can go on writing.

So, test requirement for complete path coverage for this graph with a loop will be an infinite set. So obviously, I am not going to be able to make it feasible or write a test path, set of test paths that will execute complete path coverage. Its only for completeness we really do not consider it is a useful coverage criteria by any means.

(Refer Slide Time: 23:02)



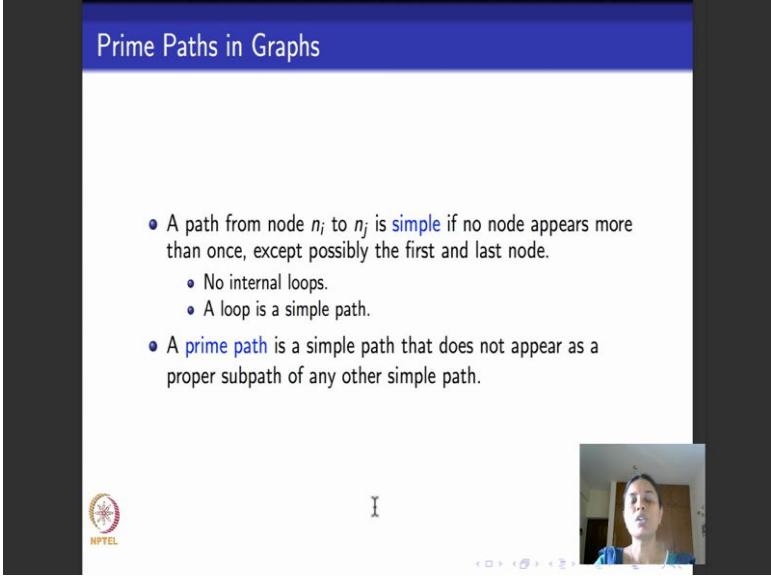
The slide is titled "Complete and Specified Path Coverage" in a blue header. It contains a bulleted list of points. The first point states that if a graph has a loop, it has an infinite number of paths, making complete path coverage infeasible. The second point asks for a good notion of specified path coverage in the presence of loops. The third point notes that loops have boundary conditions and repeated executions, and that effective test cases should not execute the loop for every iteration. The fourth point states that ideally, test cases should cover the loop, with sub-points: "Execute the loop at its boundary conditions— skip the loop execution." and "Execute the loop *once* for normal iterations." The fifth point mentions that the notion of *prime paths* came into existence for working with loops. An NPTEL logo is in the bottom left corner, and navigation icons are in the bottom right.

- If a graph contains a loop, it has an *infinite* number of paths and hence complete path coverage is infeasible.
- What will be a good notion of specified path coverage in the presence of loops?
- Loops have boundary conditions and repeated executions. Effective test cases will not execute the loop for every iteration.
- Ideally, we need to have test cases that *cover* the loop:
 - Execute the loop at its boundary conditions— skip the loop execution.
 - Execute the loop *once* for normal iterations.
- The notion of *prime paths* came into existence for working with loops.

Now we will see how to overcome this problem about complete path coverage. I have this graph with loops; the only way I have touched upon this loop is to be able to do complete path coverage. But then that is not very useful because it gives rights to an infinite number of paths. So, is there a mid way solution? So, the question that we want to ask is what will be a good notion of specified path coverage in the presence of loops. Now let us look at loops. Assuming that this kind of control flow comes from a loop, when we want to test a loop what do we ideally want to test? We say that loops have boundary conditions and then they have normal operations. So, when I test a loop maybe I want to test around its boundary condition. And let us say the loop is meant to execute 100 times, it is a far loop for I is equal to 1 to 100. So, I do not want to really execute 100 normal executions of the loop.

So, I want to be able to test the loop for its normal execution maybe once. And then I want to be able to test the loop around its boundary conditions. What happens when the loop begins, what happens when the loop ends? So, when we say we need a set off test cases that cover a loop we are looking for the following kind of test cases. When we execute the loop at its boundary conditions which will involves skipping the loop also and then we execute the loop maybe once or few number of times for its normal operations.

(Refer Slide Time: 24:36)



Prime Paths in Graphs

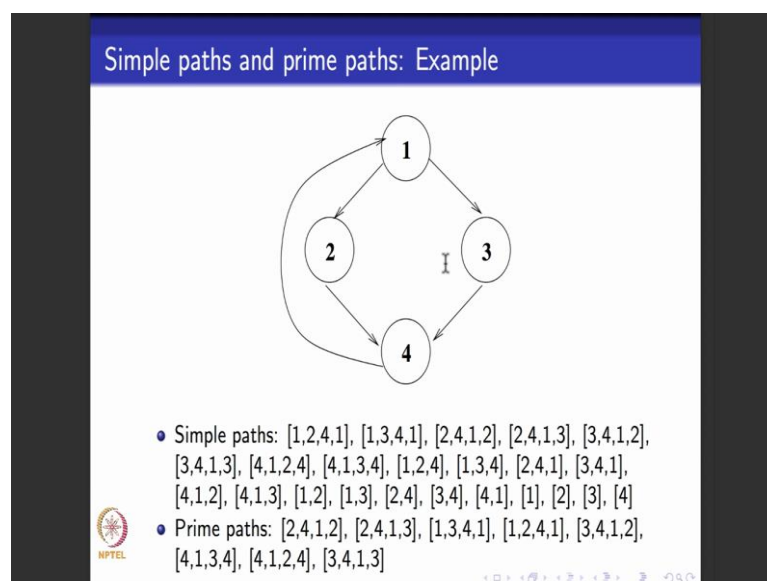
- A path from node n_i to n_j is **simple** if no node appears more than once, except possibly the first and last node.
 - No internal loops.
 - A loop is a simple path.
- A **prime path** is a simple path that does not appear as a proper subpath of any other simple path.

NPTEL

So, the notion of prime paths and prime path coverage help you to achieve this kind of execution for loops. So, what are prime paths? Before we look at prime paths I need to tell you what a simple path is. So, what is the simple path? A path from a particular node n_i to another node n_j is said to be simple, if no node appears more than once except possibly the first and the last node. So, simple path could be cycles, they could begin and end in the same node n_i and n_j could be the same vertex, but in between the path nothing appears more than once; which means there are no internal loops in the graph and every loop is believed to be a simple path.

Now, moving on ,what is a prime path? A prime path is a simple path that does not appear as a sub path of any other simple path. Just to repeat; what is a prime path? It is a simple path; and what is a second condition, second condition says that it is a simple path that does not come as a sub path of any other path.

(Refer Slide Time: 25:36)



So, I will show you an example to make this definition clear. Let us look at this graph, it has four nodes: 1, 2, 3 4. How many simple paths are there? If you go and see this listing what I have done is I have listed all the simple paths, I have listed them in a particular order in fact maybe will begin that this end. This 1, 2, 3 4 if you see right at the end of the listed simple paths they are simple paths of length 1.

Later we will move ahead and listed simple paths of lengths 2 which are basically; sorry 1, 2, 3 4 as simple paths of length 0 they just contains single vertices. Then I have gone ahead and listed simple paths of length 1 which as just edges. Then here I have listed simple paths of length 2 which are edge pairs like way, and the then I have listed simple paths of length 3.

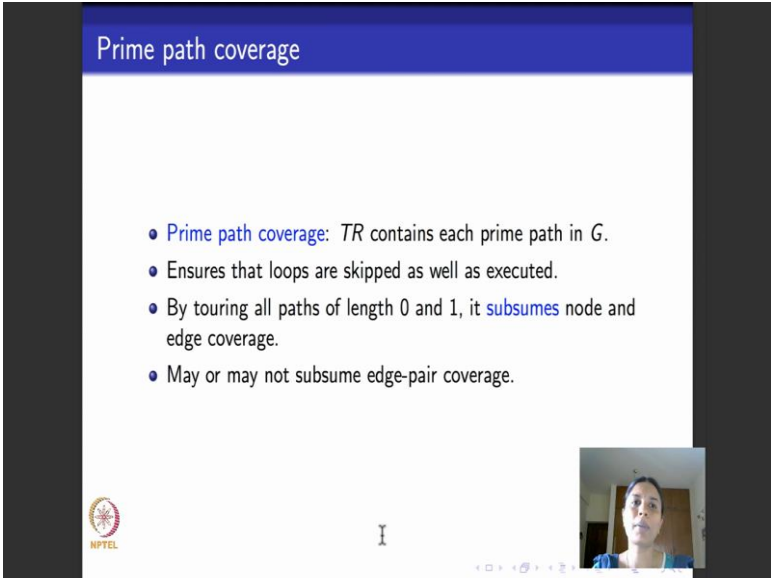
The other thing to note that this is the graph with four vertices. So, simple path means no vertex should be repeated. So, if I have a condition then what is the maximal lengths simple path that I can have with four vertices? The maximal length simple path that I can have with four vertices should be of length 3, because the moment I increase one more edge I am repeating one vertex.

So, the maximum length that I can get without repeating a vertex with four vertices is of length 3. Now if you see, if you look at all these simple paths, there so many of them, I say out of all these simple paths only these a prime paths. Why do I say these are prime paths? Go back to that definition of prime- path prime path should be simple, prime path

should not be a sub path of any other path. So, if you take a path that looks like this let us take 4, 1, 2 this path; 4, 1, 2. If I take a path like that this 4, 1, 2 path comes as a sub path of this path here 2, 4, 1, 2 it also comes as a sub path of this path here 3, 4, 1, 2. So, 4, 1, 2 does not qualified to be a prime path.

Similarly, if I take something like 2, 4; 2, 4 occurs a sub path here 1, 2, 4, it occurs a sub path here 2, 4, 1 it occurs as a sub path here, here, here, several places. So, path like 2, 4 does not qualified to be a prime path. If I go on looking like this, I basically eliminate all paths of length 0, 1, 2 and 3. And only paths of length four in this case happened to be prime paths, because none of these paths of length four occur as sub paths of each other. So, this graph has several simple paths, but it has only about eight prime paths.

(Refer Slide Time: 28:09)



Prime path coverage

- **Prime path coverage:** TR contains each prime path in G .
- Ensures that loops are skipped as well as executed.
- By touring all paths of length 0 and 1, it **subsumes** node and edge coverage.
- May or may not subsume edge-pair coverage.

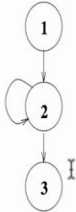
Now, what is prime path coverage? Prime path coverage very simple; the test requirement for prime path coverage says- please cover all the prime paths in G . Now, to be able to meet this test requirement you need to write a set of test paths that first identify the prime paths in C and then cover them. So, in the next lecture I will tell you how to do this, what are the algorithms that will do this, but for now we will go ahead and see a little bit more about prime paths coverage without worrying about how to do it.

So, the first thing that I want to tell you is that prime path coverage actually meets this loop coverage criteria that I told you about; this one test a loop at its boundary and test a loop for its normal operations. So, we will understand how that happens. Prime path


coverage ensures that loops are skipped and executed. And towards all paths of length 0 and 1 we saw that for that example at least. So, by default it subsumes node and edge coverage, because if you see here it includes node coverage and it includes edge coverage.

(Refer Slide Time: 29:23)

Prime path coverage vs. edge-pair coverage



- In graphs where there are self loops, edge-pair coverage requires the self loop to be visited.
- For e.g., in the above graph, TR for edge-pair coverage will be $\{[1, 2, 3], [1, 2, 2], [2, 2, 3], [2, 2, 2]\}$.
- Some of these are prime paths/simple paths.
- TR for prime path coverage for the above example is $\{1, 2, 3\}, [2, 2]\}$.



This node says that it may or may not subsume edge pair coverage. Why is that so? I will show you an example: considered a graph that looks like this, it has three vertices and then it has got a self loop here. So, edge pair coverage for this graph requires that the self loop needs to be visited. So, edge pair coverage for this graph we will say you visit this edge 1, 2, 2; you visit this path of lengths 2 which is 1 to 2 and 2 to itself. But if you look at this path 1 to 2; if you see the vertex 2 repeats here. And that violates the condition of it being a prime path. Remember prime path a simple paths no intermediate vertices as supposed to repeat then non supposed to have loops.

So, except for this problem prime path coverage does cover edge pair coverage, but not for graphs like this; if a graph looks like this right prime path coverage does cover edge pair coverage. But if a graph has a self loop then prime path coverage does not cover edge pair coverage.

(Refer Slide Time: 30:19)

Prime path coverage and loops in graphs

Prime paths capture the notion of *covering* a loop well.

- There are nine prime paths.
- They correspond to
 - 1,3,5,7 : Skipping the loop,
 - 1,3,5,6 : Executing the loop once, and
 - 6,5,6 : Executing the loop more than once.

NPTEL

So, now we will move on and understand how prime paths cover the notation of a loop. So, we will go back to this example graph that we looked at a few slides earlier. In this example there is this loop here between nodes 5 and 6. And I am not listing the prime paths for this graph, but you can take it on faith that this graph has nine prime paths. In the next module I will tell you how to list all those nine prime paths.

So, then if we see this is one prime path; 1, 3, 5, 7 because it is a simple path and it does not occur as a sub path of any other path. So, this prime path for this particular example corresponds to skipping this loop, because it completely avoids this loop and if you see here is another prime path 1, 3, 5, 6. This prime path gets into the loop, it corresponds to executing the loop once.

And then this is another prime path for this example 6, 5, 6. Why is this prime path? Remember in prime paths which have simple path the beginning and ending nodes are allowed to repeat, only the intermediate nodes are not allowed to repeat. So, 6, 5, 6 is a prime path. And, what is the main job then 6, 5, 6 is doing for this graph? It is executing the loop more than once. So, if we see these three prime paths 1, 3, 5, 7 skips the loop 1, 3, 5, 6 enters the loop and then 6, 5, 6 helps you to execute the loop for its normal operations. So, it is intuitively in this sense that prime paths exactly capture loop coverage the way we want them to do in test case design.

(Refer Slide Time: 32:04)

Implementing test requirements for prime path coverage

- Prime paths, by definition, do not have internal loops.
- In many cases, it might be impossible to meet the test requirement of prime path coverage without internal loops.
- That is, test paths that meet prime path coverage TR need to have internal loops to make prime path coverage feasible.

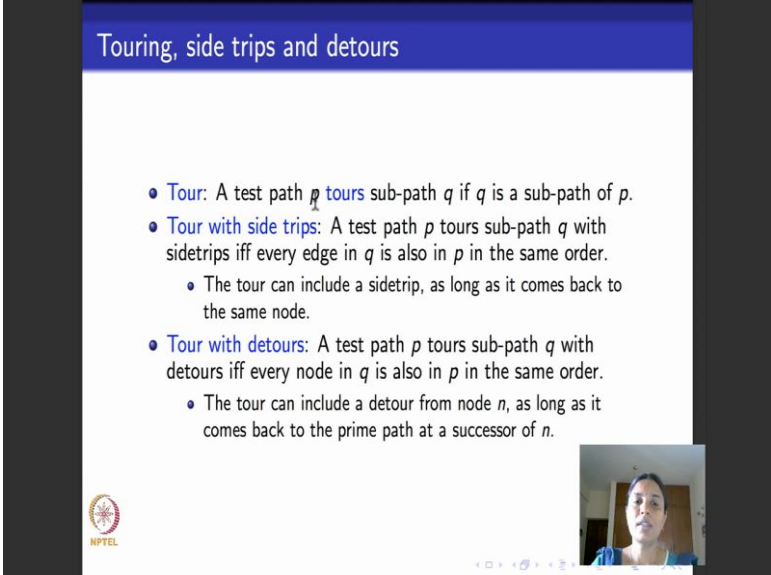
NPTEL

So, one important thing is that, I told you right, in the next module I tell you how to come up with an algorithm that will help us to design test cases that will need the test requirement of prime path coverage. But before we do that I will tell you we could have some problems. Like for example, you take a graph like this. Prime path coverage for this graph, this is a prime path 1, 2, 3, 4, 5, this is a prime path. It might so happened this graph if this graph corresponds to the control flow graphs is some piece of code here is a loop. It might so happen that this code from which this control flow graph is derived, the loop present at statement number 3 and 6 is such that you can never write you have to execute the loop at least once.

Like for example, if I take a C program. There are two kinds of loops: there is a do-while and a while-do loop right. While-do loop first checks the condition and then executes the loop, but if I have a do-while loop I have to execute the loop at least once before I move on. So, it might be an infeasible test requirement for that kind of a code when I say that you achieve this prime path without executing the loop. Why should I not execute the loop? Because if I execute the loop then the vertex three which is an intermediate vertex in this path will occur more than once. So, it will not be a prime path. So, 1, 2, 3, 6, 3, 4, 5 is not a prime path because 3 occurs more than once; 1, 2, 3, 4, 5 is a prime path. But to be able to write a test path and execute it in the code to achieve 1, 2, 3, 4, 5 the code might be such that I might have to go through this loop once. Like I told you write the code might have a do while statement.

So, how do we get over this? We get over this by using a notion of a side trip and a detour. So, we will see what they are.

(Refer Slide Time: 34:09)

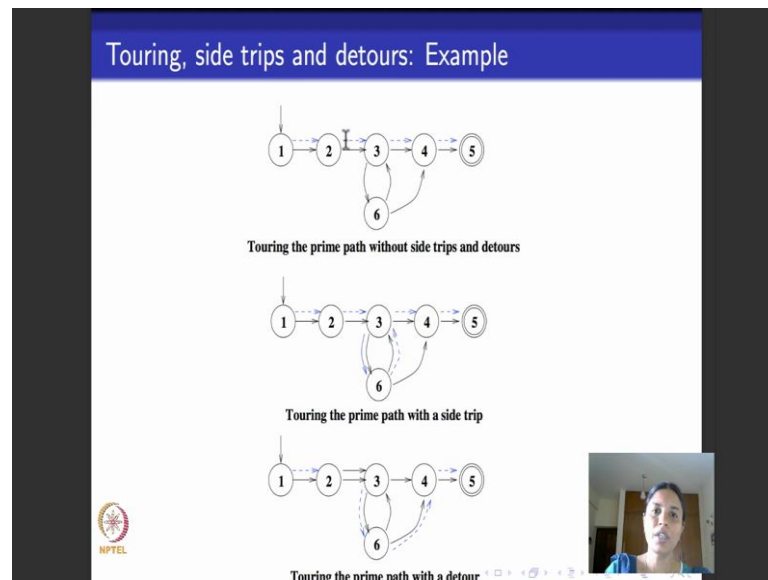


The slide is titled "Touring, side trips and detours" in a blue header. It contains three bullet points defining different types of tours. In the bottom right corner, there is a small inset video of a woman speaking. The NPTEL logo is in the bottom left corner.

- **Tour:** A test path p **tours** sub-path q if q is a sub-path of p .
- **Tour with side trips:** A test path p **tours** sub-path q with sidetrips iff every edge in q is also in p in the same order.
 - The tour can include a sidetrip, as long as it comes back to the same node.
- **Tour with detours:** A test path p **tours** sub-path q with detours iff every node in q is also in p in the same order.
 - The tour can include a detour from node n , as long as it comes back to the prime path at a successor of n .

So, what is a tour? Tour I introduced you in the last module when we looked at graphs tour is a test path that tours is a sub path if q is a sub path of the overall test path. So, what is a tour with a side trip? A tour with a side trip is the following--- test path p towards the sub path q with side trips, if every edge of q is also in p in the same order. It is like taking a side trip as a part of a main holiday. And what is a tour with the detour? A tour with a detour is the test path p towards a sub path q with the detour if edges do not come in the same order, but vertices come in the same order.

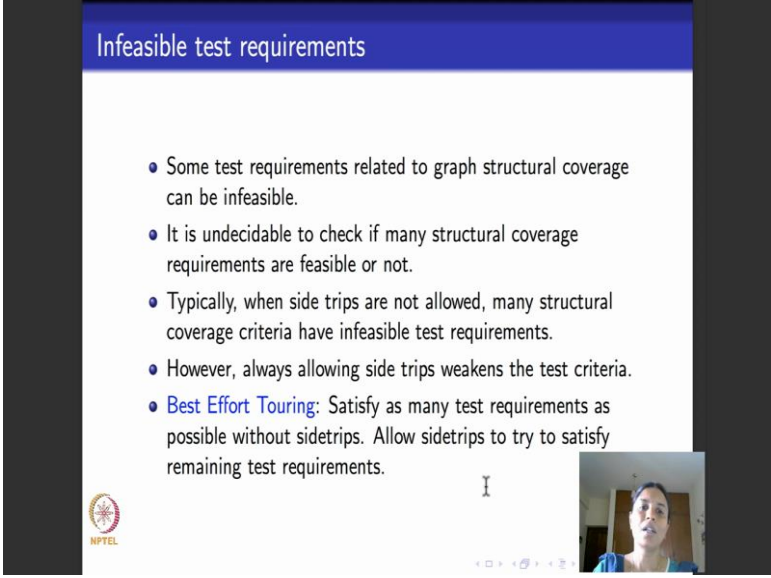
(Refer Slide Time: 34:52)



So, I will show you an example. So, here is my test path 1, 2, 3, 4, 5. I take the same test path to meet the prime path coverage of 1, 2, 3, 4, 5 with the side trip. So, what do I do in the side trip? Side trip says that it is a same path 1, 2, 3, 4, 5, but I do 1, 2, 3 take a side trip around 6 come back to 3 and do 4, 5. So, I retain the vertices that come in the main path. See, side trips says that you retain vertices that come in the same path in the same order; retain the vertices 1, 2, 3, 6, 3, 4, 5, but in the main path which is this blue dotted line the vertices occur in the same module.

Now what is a detour? Detour says you retain edges in the same order, may or may not retained vertices in the same order. So, a detour looks like this 1, 2, 3, 6, 4, 5. It has left this edge it does not retained edges in the same order, but it retains this path. So, why do I need this? I need this because to be able to achieve this prime path coverage I might have to go through the loop. And if I go through the loop the prime paths ceases to be at prime path. So, I do a small work around then say I actually cover this prime path, but with the help of a side trip. Sometimes I might say I cover this prime path, but with the help of a detour. They have just extra additions that we add so as to not to alter the notion of prime paths being simple paths, but I still want to be able to achieve these test requirements.

(Refer Slide Time: 36:30)



The slide is titled "Infeasible test requirements" in a blue header. It contains a list of five bullet points. The first four are in blue, and the fifth is in black. A small inset video in the bottom right corner shows a woman speaking. The NPTEL logo is in the bottom left corner.

- Some test requirements related to graph structural coverage can be infeasible.
- It is undecidable to check if many structural coverage requirements are feasible or not.
- Typically, when side trips are not allowed, many structural coverage criteria have infeasible test requirements.
- However, always allowing side trips weakens the test criteria.
- **Best Effort Touring:** Satisfy as many test requirements as possible without sidetrips. Allow sidetrips to try to satisfy remaining test requirements.

So, this is what I was telling you about. Some test requirements related to graph coverage criteria may be infeasible. In fact, it is undecidable to check whether even test requirement is feasible or not, we won't really go onto its details. But typically when I allow side trips, then I might be able to achieve test requirements with reference to a particular code.

So, what is called a best effort touring is that you tried to satisfy as many test requirements is possible without side trips or detours, and then if you still have test requirements that are unachievable, consider using side trips and detours to be able to satisfy them.

(Refer Slide Time: 37:09)

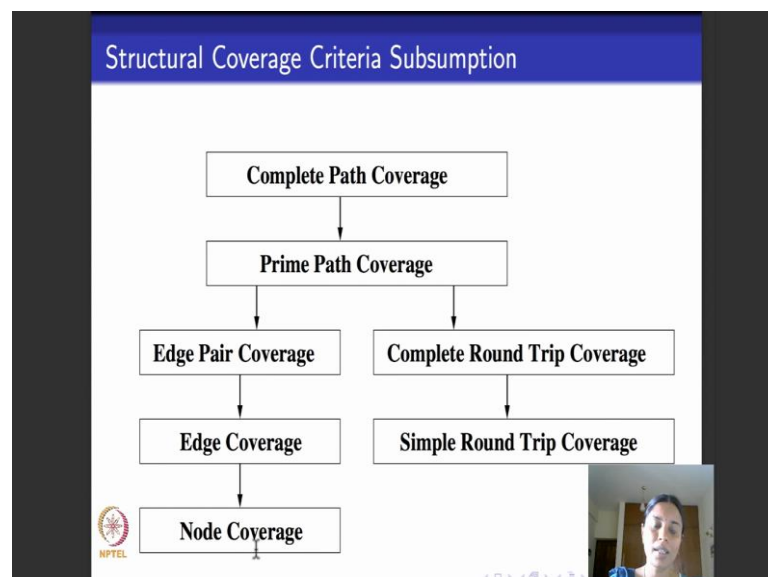
Round trips

- **Round trip path:** A prime path that starts and ends at the same node.
- **Simple round trip coverage:** TR contains at least one round trip path for each reachable node in G that begins and ends in a round trip path.
- **Complete round trip coverage:** TR contains all round trip paths for each reachable node in G .
- The above two criteria omit nodes and edges that are not in round trips.
- Hence, they do not subsume edge-pair, edge or node coverage.

NPTEL

So, what is a round trip? A round trip is a prime path that starts and ends in the same node, we have seen these, these suggest specific kinds of prime path. What is a simple round trip coverage? Test requirement contains one round trip path for each reachable node that begins and ends in the same vertex. What is complete round trip coverage? Test requirement contains all the round trip paths. So, they subsume edge and node; I mean they do not subsume edge pair edge or node coverage.

(Refer Slide Time: 37:42)



Now to summarize what are the various coverage criteria we saw; we saw node coverage, visit every node; edge coverage, visit every edge; edge pair coverage visit every path of length at most two, visit prime paths means compute on the prime paths right test paths that exactly visit every prime paths. We will see how to do this in the next module. Then, complete path coverage which means visit every path in the graph which is, I believe, a useless test requirement. And then we had these two which talk about round trips. They are basically prime paths, but begin and end with the same node.

And why have I put them in the structure? This structure captures how each of them subsume the other. We discussed this right, edge coverage subsumes node coverage. Edge pair coverage subsumes both edge coverage and node coverage. Prime path coverage, except for graphs with self loops, subsume edge pair coverage, edge coverage and node coverage. Complete path coverage subsumes everything, but is infeasible and useless.

So, what we will see in the next module is how do we look at algorithms that given each of these structural coverage test requirements, how do I come up with the test cases? Edge, edge pair and node coverage a easy to do, but prime path coverage is a nice algorithm. So, we will spend most of the next module looking at these algorithms.

Thank you.

Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

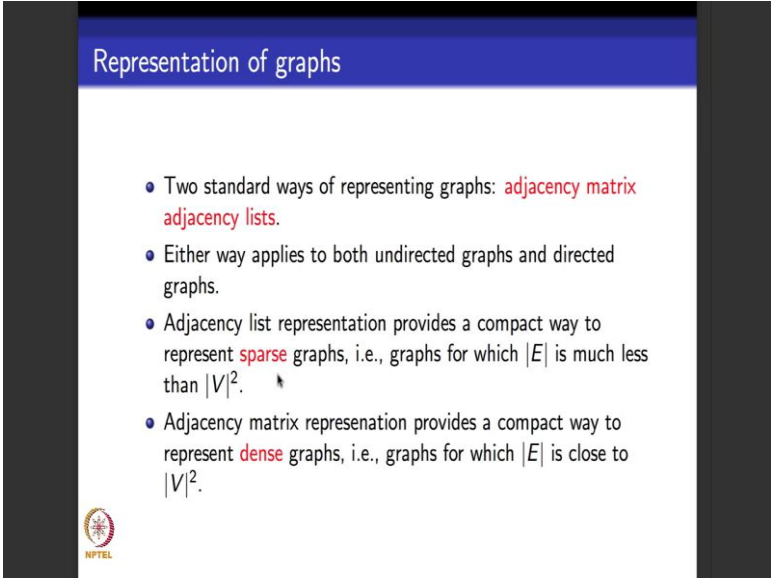
Lecture - 07
Elementary Graph Algorithms

Hello everyone. Welcome to the next module. In this module what I would like to spend some time on is relating to elementary graph algorithms. The last module we saw structure in coverage criteria related to graphs, we saw how to rectify test requirements for various structural coverage criteria; we began with node coverage, edge coverage, went on till we saw prime path coverage.

The next thing that I would like to focus on as far as testing is concerned is how to define algorithms that will help us to write test requirements and test paths that we achieve various graph coverage criteria. It so turns out the algorithms that will help us to write test paths and test requirements for graph coverage criteria use elementary graph search algorithms; most of them use in fact BFS- breadth first search. Some of them can also use things like finding strongly correcting components, finding connected components etcetera. So, once you know these algorithms very well it is a breeze to be able to work with algorithms that deal with structural coverage criteria and graphs.


So, what I am trying to spend time on in this module and in the next module is to help you to revise elementary graph search algorithms. We will look at breadth search first search in this module, in the next module we will recap depth for search and also look at algorithms based on DFS that will help us to output strongly connected components. Once you know these algorithms you will move on come back and look at the algorithms to write test requirements in test paths for structural coverage criteria based on BFS and DFS.

(Refer Slide Time: 01:59)



Representation of graphs

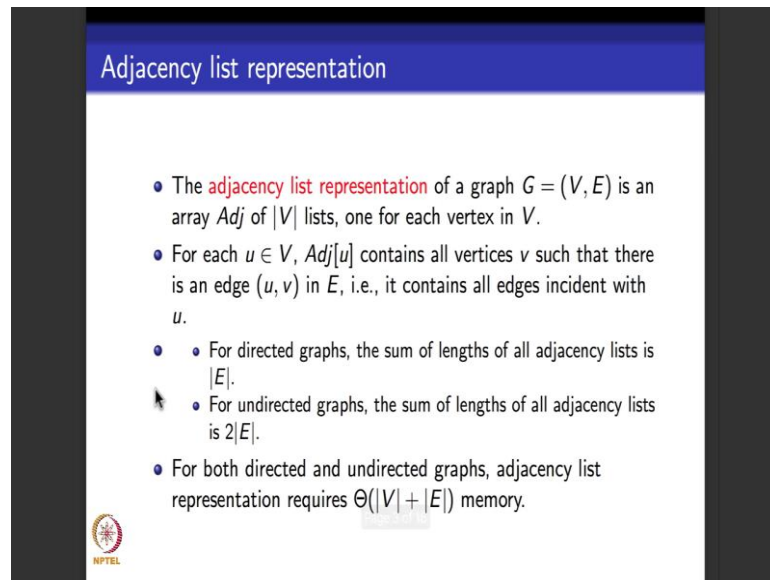
- Two standard ways of representing graphs: **adjacency matrix** **adjacency lists**.
- Either way applies to both undirected graphs and directed graphs.
- Adjacency list representation provides a compact way to represent **sparse** graphs, i.e., graphs for which $|E|$ is much less than $|V|^2$.
- Adjacency matrix representation provides a compact way to represent **dense** graphs, i.e., graphs for which $|E|$ is close to $|V|^2$.



Let us begin looking at what are the standard representations of graphs that algorithms that manipulate graphs will look. Like graphs can be represented in two standard ways as you might know; as an adjacency list and adjacency matrix. Both undirected and directed graphs can be represented both as an adjacency list and as adjacency matrix. It so turns out that in testing when we look at graphs as data structures modeling various software artifacts, we will never look at undirected graphs they are not very useful to us maybe except rarely when we look at let us say class graph or things like that otherwise most of the graphs that we will deal with will be directed graphs.

But why I am saying this that the algorithms that we will look in this module and next module work both well for directed graphs and for undirected graphs. We will use then for directed graphs. So, we will go ahead and see what an adjacency lists looks like.

(Refer Slide Time: 03:01)



The slide is titled "Adjacency list representation" in a blue header. It contains a list of bullet points explaining the concept. The first bullet point states that the adjacency list representation of a graph $G = (V, E)$ is an array Adj of $|V|$ lists, one for each vertex in V . The second bullet point explains that for each $u \in V$, $Adj[u]$ contains all vertices v such that there is an edge (u, v) in E , i.e., it contains all edges incident with u . The third bullet point has two sub-bullets: for directed graphs, the sum of lengths of all adjacency lists is $|E|$; for undirected graphs, the sum of lengths of all adjacency lists is $2|E|$. The fourth bullet point states that for both directed and undirected graphs, adjacency list representation requires $\Theta(|V| + |E|)$ memory. There is a small logo in the bottom left corner of the slide.

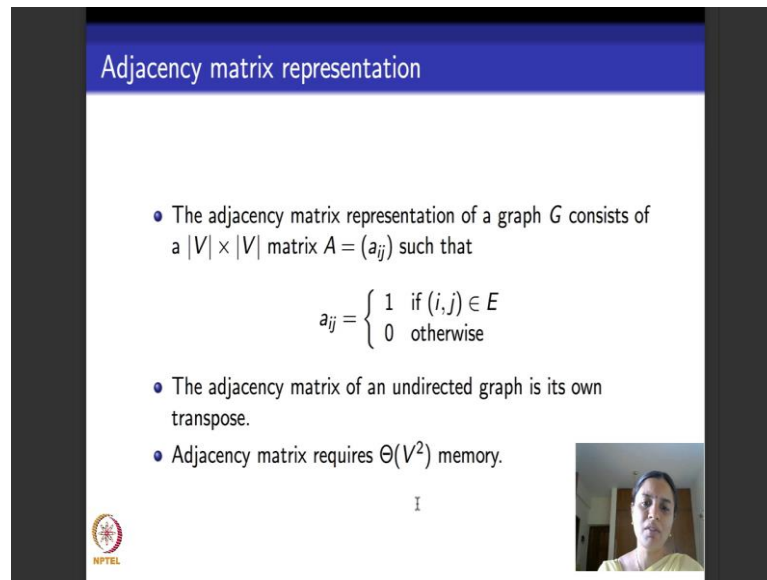
- The **adjacency list representation** of a graph $G = (V, E)$ is an array Adj of $|V|$ lists, one for each vertex in V .
- For each $u \in V$, $Adj[u]$ contains all vertices v such that there is an edge (u, v) in E , i.e., it contains all edges incident with u .
- For directed graphs, the sum of lengths of all adjacency lists is $|E|$.
- For undirected graphs, the sum of lengths of all adjacency lists is $2|E|$.
- For both directed and undirected graphs, adjacency list representation requires $\Theta(|V| + |E|)$ memory.

So, given a graph with vertex at V and at set E , what is an adjacency list? An adjacency lists keep an array of lists. How many lists are there in the array? There is one list for every vertex, so totally there are mod V lists. What do each of this lists contain each of this lists contain all the vertices that are adjacent to the given vertex v ; that is it contains each vertex u such that (u, v) is an edge in the graph. So, what does and adjacency lists contain? It contains an array of mod V list one list for every vertex where the list for every vertex contains all the other vertices that are connected to this vertex through an edge.

What is the size of an adjacency list? If you see for directed graphs the directions of the edge do not matter right. So, when I have an edge (u, v) I practically have two edges (u, v) and (v, u) . So, for undirected graphs the size of the adjacency lists is two times model: one edge two edges for every edge present in the graph in the adjacency list. And for directed graphs there is one for every edge there is one vertex somewhere in the adjacency list of some vertex, so the size of adjacency list is mod E .

So, for both directed and undirected graphs this total size of the adjacency list will be this. This first mod V in the theta component represents the number of lists and each list can be at most mod E size. So, the total size is theta of mod V plus mod E .



(Refer Slide Time: 04:40)



Adjacency matrix representation

- The adjacency matrix representation of a graph G consists of a $|V| \times |V|$ matrix $A = (a_{ij})$ such that
$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \\ 0 & \text{otherwise} \end{cases}$$
- The adjacency matrix of an undirected graph is its own transpose.
- Adjacency matrix requires $\Theta(V^2)$ memory.

I



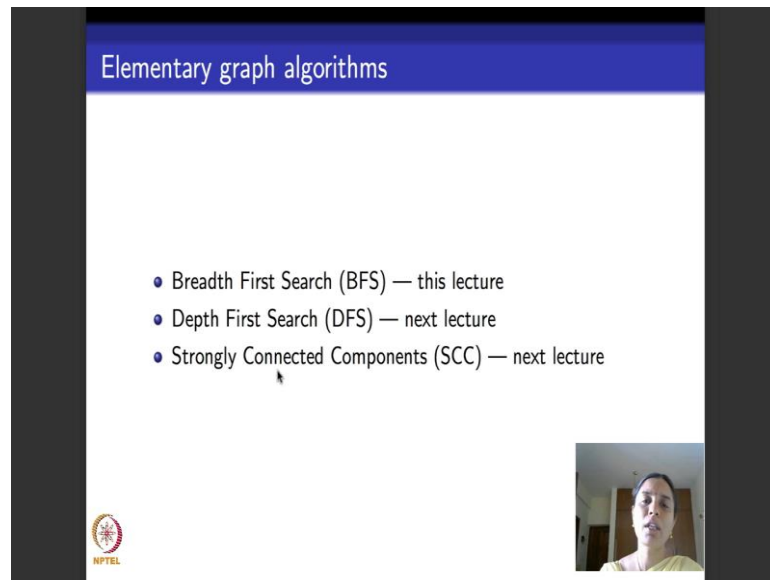
Moving on what is an adjacency matrix representation of a graph look like? Adjacency matrix of a graph is a mod V by mod V matrix. So, it has as many entries is that of number of vertices, it is a square matrix, and its filled with 0 and 1 entries. It contains a 1 at vertex v_i and v_j if the edge (v_i, v_j) is in the set of edges of the graph otherwise it contains a 0. So, if you see adjacency matrix will go directed and undirected graphs will need V squared, theta V squared memory, because they are of matrices of size mod V by mod V . In addition for undirected graph because the edges are there on both sides, the transpose of its adjacency matrix will be same as the given adjacency matrix.

So, which is good for what kind of graphs? Graphs could be dense or they could be sparse. If a graph sparse means it has very few edges; it has lot of vertices which has very few edges. Trees are examples of graph that has sparse. So, graphs are sparse then adjacency list is considered to be a good representation, because the size of each list for every vertex will be small. On the other hand if a graph is dense, which means it has lot of edges the number of edges is close to mod V squared, then an adjacency matrix is considered good, because the size of the adjacency matrix is fixed to be V by V matrix.

So, if there are many edges it is just means more one entry lesser 0 entries, whereas if it is an adjacency list for a dense graph then the size of the each list would be really long. So, for sparse graph adjacency lists are considered to be good, for dense graph adjacency matrices are considered to be good; otherwise there is no really big trade of about one

representation or the other it is just for the convenience of our this thing. In our algorithms we will mainly use adjacency lists. The algorithms that we will deal with in this two modules we will use adjacency lists.

(Refer Slide Time: 06:41)

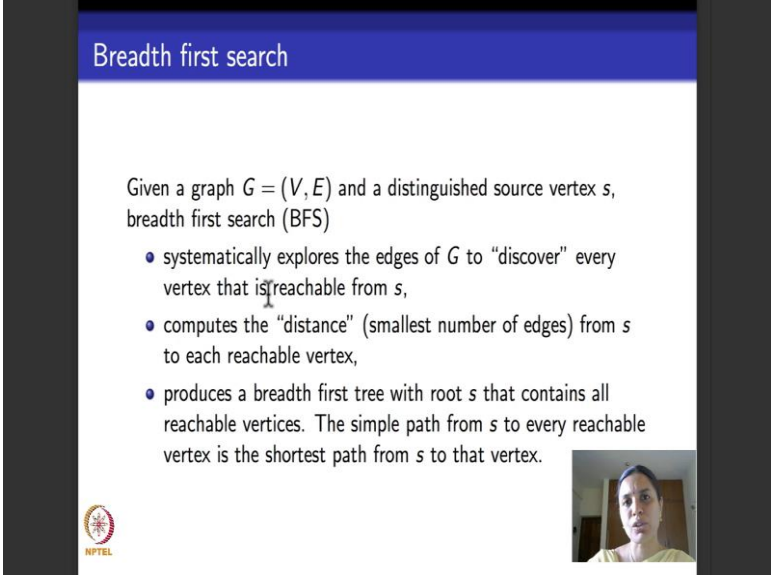
A screenshot of a presentation slide titled "Elementary graph algorithms" in a blue header. The slide lists three topics: "Breadth First Search (BFS) — this lecture", "Depth First Search (DFS) — next lecture", and "Strongly Connected Components (SCC) — next lecture". Each item is preceded by a blue circular bullet point. In the bottom left corner, there is a small NPTEL logo. In the bottom right corner, there is a small video inset showing a person speaking. The slide is framed by dark vertical bars on the left and right sides.

Elementary graph algorithms

- Breadth First Search (BFS) — this lecture
- Depth First Search (DFS) — next lecture
- Strongly Connected Components (SCC) — next lecture

So, what are the elementary graphs such algorithms that we will be looking at? We look at three algorithms, because these three are the main ones that we will need for test case generation. So, we will look at breadth first search which we will do in this module. The next module I will help you recap depth first search and we will also depth first search and breadth for search have several applications. So, we look at one particular algorithm which deals with how to compute strongly connected components or connected components using depth first search. So, those two we will look at it in the next lecture.



(Refer Slide Time: 07:12)



Breadth first search

Given a graph $G = (V, E)$ and a distinguished source vertex s , breadth first search (BFS)

- systematically explores the edges of G to “discover” every vertex that is reachable from s ,
- computes the “distance” (smallest number of edges) from s to each reachable vertex,
- produces a breadth first tree with root s that contains all reachable vertices. The simple path from s to every reachable vertex is the shortest path from s to that vertex.

So, we will move on and start with breadth first search. What is breadth first search do? As the name says it searches through a graph, and how does it search through a graph? It searches through a graph in a breadth first way. That is it starts its search from a particular vertex, let us say call it source vertex and what does it do is that it first explores the adjacency list of the source vertex; that is, it explores the span of the breadth of the graph at the vertex s . Once it is finished exploring the adjacency list of the vertex s which is the source, what it means to explore, it adds it to a particular queue that it gives, it goes ahead takes one vertex at a time and explores that vertex adjacency lists and it goes on like this.

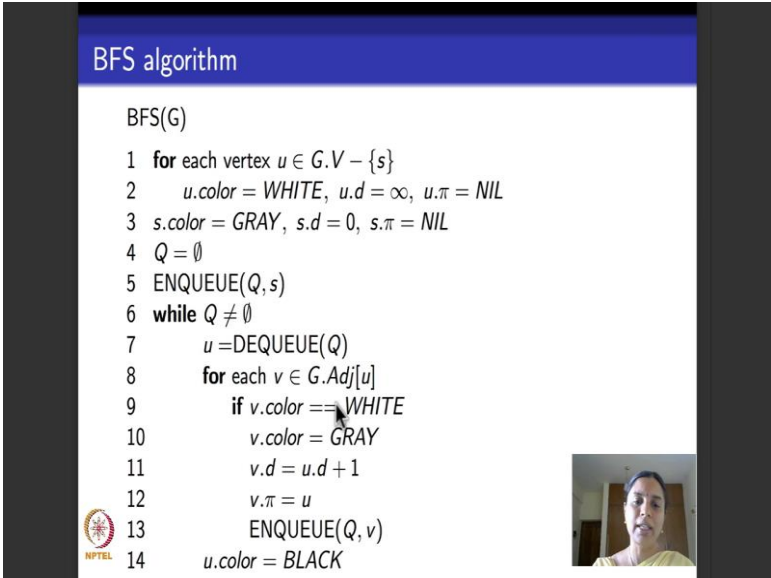
So, this way of searching or traversing through a graph by exploring the adjacency list of each vertex fully is called breadth first search, because it explores the graph in a breadth first way; does not go deep unlike depth first search. As it explores it also computes a few things which are useful for us to keep. What it says is as it explores it also keeps track of given a fixed source s what is the distance of any other vertex in the graph from the source s .

If we look at the graphs that do not have any attributes like weights rather things attached to its edges. So, for us distance plainly means the number of edges. So, how far is the particular vertex in the graph from the designated source vertex? How far means how

many edges are there between the particular vertex in the graph that is reachable from the source vertex. That is what is called the distance of a vertex from its source.

Breadth search first also computes the distance. It so happens breadth first search computes this smallest distance or the shortest distance we will see what is that in a meanwhile. And it once you explore a graph by using breadth first search then the result of that exploration is a tree containing the path from the source to call other vertices that are reachable from the source. Such a tree is called breadth first tree. Breadth first search algorithm can be used to output the shortest distance of every vertex from the source. It can also be used to output the breadth first tree which contains the shortest path from the source to every other vertex.

(Refer Slide Time: 09:32)



BFS algorithm

```
BFS(G)
1  for each vertex  $u \in G.V - \{s\}$ 
2     $u.color = WHITE, u.d = \infty, u.\pi = NIL$ 
3   $s.color = GRAY, s.d = 0, s.\pi = NIL$ 
4   $Q = \emptyset$ 
5  ENQUEUE( $Q, s$ )
6  while  $Q \neq \emptyset$ 
7     $u = DEQUEUE(Q)$ 
8    for each  $v \in G.Adj[u]$ 
9      if  $v.color == WHITE$ 
10        $v.color = GRAY$ 
11        $v.d = u.d + 1$ 
12        $v.\pi = u$ 
13       ENQUEUE( $Q, v$ )
14   $u.color = BLACK$ 
```

So, here is how the algorithm looks like; before I show you the pseudo code of the algorithm I will tell you what it does.

(Refer Slide Time: 09:40)

BFS: Queue of vertices

- BFS colours each vertex with one of the three colours: white, gray or black.
- To start with a vertex is white, it is not yet discovered and is not in the queue Q.
- When it is discovered but not fully explored, it is gray and gets into the queue Q.
- Its color is black when it is fully explored and is out of the queue Q.

NPTEL

So, it takes the graph and the main job that breadth first search does is to be able to color each vertex in the graph; keeps the queue of vertices and that queue contains the queue of vertices that are colored grey. So, what is breadth first search do? It keep it color codes every vertex and there are three kinds of colors that it keeps. To start with every vertex is colored white. And then vertex that is colored white further changes to color grey and after its colored grey it changes to color black.

Once it changes to color black we say we are done with exploring that vertex fully. So, to start with all vertices are white, and when does the white vertex become grey? It becomes grey, when it enters the breadth first search maintains a queue of vertices and when it enters queue it becomes grey. So, it becomes grey when it is first discovered as a part of the adjacency list of some vertex. And then it is put into the queue.

When it is put into the queue the aim is grey colored it is discovered now, but I am yet explore this vertex's adjacency list. So, I have put it into queue to remember that I have to do that. And I move on and explore the adjacency list of all other siblings of this particular vertex. And then when I come back and explore this particular vertex's adjacency list and finish exploring that then I color it black, and when I color it black I will move it form the queue.

So, how does the breadth first search work? So, here is what is the pseudo code for breadth first search. So for each vertex that is not the source, remember, s is the source

for which breadth first search algorithm start. So, for each vertex read $G \cdot V_s$ for each vertex in the graph that is not the source which is not s set its color. So, we keep three attributes with each vertex; we keep a color attribute which tells you what the color of the vertex is: it could be white, grey or black. We keep a d attribute, d for distance which tells you what is the distance in terms of the number of edges of this vertex from the source vertex s . And we keep a p_i attribute: p_i representing predecessor of parent which tells you who is the parent of a particular vertex in the breadth first tree that is algorithm is going to output.

So, what are the three attributes that we keep with every vertex? A color attribute, a distance attribute which tells its distance in terms of a number of edges from the source, and a predecessor or a p_i attribute which tells you who is the predecessor of this particular vertex in the breadth first tree.

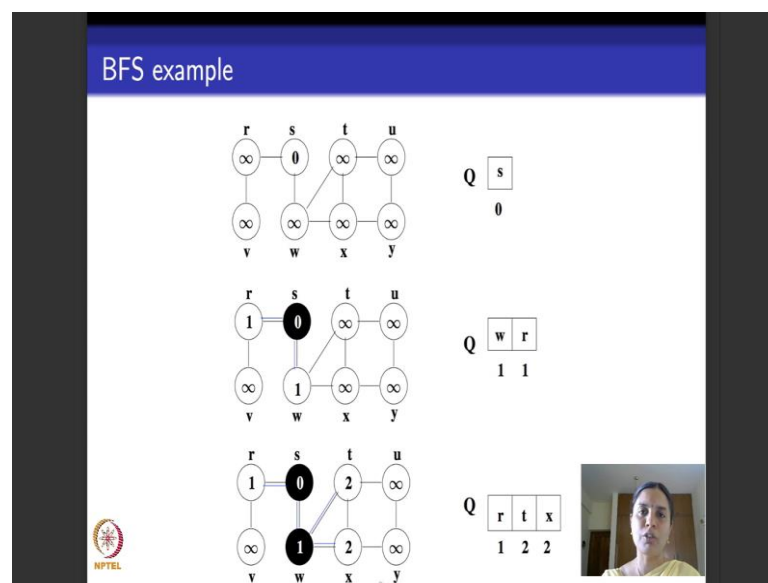
So, to start with breadth first search algorithm colors each vertex white, sets its distance to be infinity, because the distance will reduce right, to start with I do not know how far it is from the source. And if the tree is not yet developed so the breadth first search algorithm is just began, so its p_i attribute is NIL. Now it begins its search from the source vertex s . So, the first thing that gets into the queue; Q is the Q queue it maintains. The first vertex that gets into the queue is s . So, if you see this enqueue Q s puts s inside the queue Q . So, when puts inside the s Q the color of the vertex s is grey; s the distance from s itself is 0, s is its own parent so it does not have predecessor, its predecessor is set to NIL. So, Q is initialized and s is put into the queue.

So, as long as there are vertices in the queue which means as long as there are grey color in the vertices what do you do? You pick up the first vertex available from the Q , you dequeue it right and you start exploring adjacency list that is what this one for loop line number 8 does. So, it says for each other vertex in the adjacency list of the vertex u in the graph. If its color is white then you make it grey, which means you have discovered that vertex. Now you set its distance from the source s to be whatever the distance from the source s of u was plus 1. And then you say its parent in the breadth first search tree is going to be u , because I found it as the part of the adjacency list of u so it is natural that its parent is u .

Once you do this, you add V to the Q that you maintain. And you keep doing this, keep doing this, till you finish exploring the adjacency list of u . Once you fully finish exploring the adjacency list of u , you come out and color u as black. So is it clear what the algorithm does? Just to quickly repeat. To start with the color each vertex white sets the distance and bi attributes, it begins its search at the source vertex s , add this to the queue, colors it grey, sets it g and pi attributes, and then for each vertex in the Q it takes it out from the Q explores the adjacency list of the vertex that was just taken out fully.

What does it mean to explore the adjacency list of the vertex? It looks at each vertex in the adjacency list of the vertex u , if it was colored white, which means it was not yet discovered it says I am discovering it now marks it grey; sets its distance from s as the distance of u from s which was already set plus 1; sets its parent in the breadth first search tree to be u and adds it to the Q . When its finish doing this for all the three adjacency list of u , it removes u from the Q here and colors it black.

(Refer Slide Time: 15:12)



So, here is an illustration of how breadth first search works on a small example. In this particular case I have taken an undirected graph as a example, but as I told you in the beginning directed or undirected graph does not matter. The algorithm will work fine because it just takes an adjacency list as an input and does not really worry itself about whether the graph is directed or undirected.

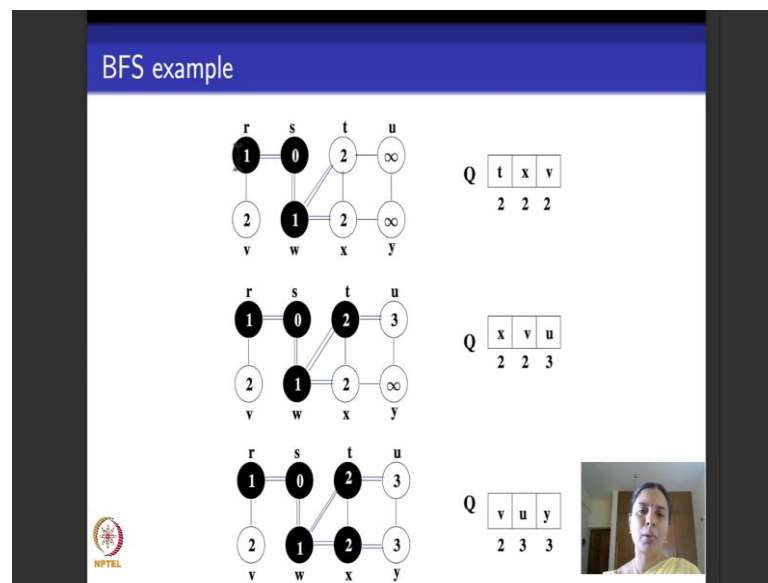
So, there are eight vertices in the graph; the vertex s is source vertex. So, what are these various things? The other vertices are r, s, t, u, v, w, x, y . So, I have also put these parameters inside the vertex, what they correspond to is the d attributes. You see the distance from s , the source to itself, is 0 to start with every other vertex is this thing infinity from the source s . The Q initially has s its distance from itself is 0.

Now I start exploring the adjacency list of s because that is what is there in my queue. So, what are the vertices that are adjacent to s ? That is r and w . So, this blue color arrow, I hope you can see them means that I have set the predecessor attribute. So, I have set the predecessor of r as s and I have set the predecessor of w also as s . So, so far my breadth first tree that I output which I read out from the predecessor attributes contains this sub graph; it contains the node s with its successor as r and another successor as w . And because I have discovered these two vertices for the first time I have put them in the Q and set their distances from the set s to be one because they are reachable from s through one edge.

Now, you can put them in any order. It just so happens in this case that I put w first and then r , nothing will go wrong in algorithm if you put r first and then w . Now because I have put w first I start exploring the adjacency list of w . What are the two vertices that are adjacent to w ? If you look at the third figure here, t and x . So, I add them to the breadth first tree which is again setting its pi attributes. So, I color this as blue indicating that the predecessor of t is w and then I color this edge also blue indicating that the predecessor of x is also w and I add t and x to the Q . And how far are they from the source s ? They are at distance two from the source s right because they there are two edges you take two edges to reach t .

Please remember that I had not colored vertices grey in this example, because it became a little cumbersome to do it. But what all the vertices are there in the Q are all colored grey, but it is not depicted in the figure here.

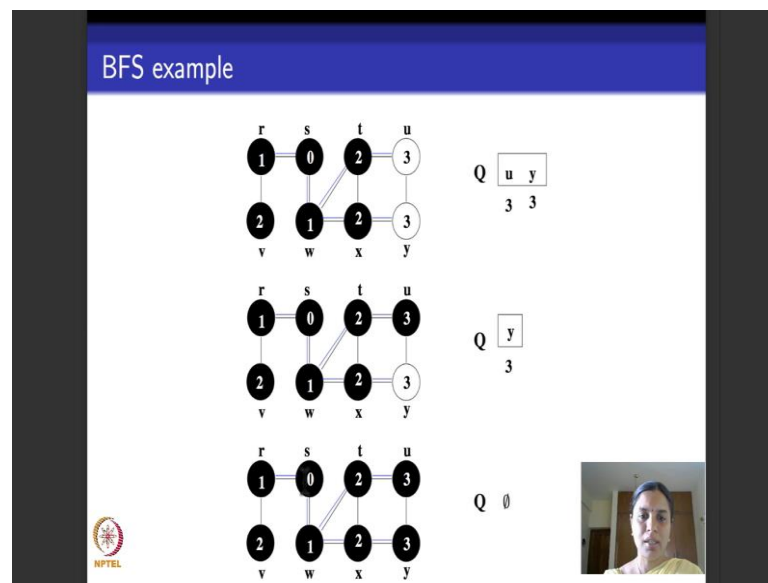
(Refer Slide Time: 18:00)



And move on like this: now I start exploring the adjacency list of vertex r because that is what is there at the head of the Q . So, V is adjacent to r , so add V to the Q here, color r black and come out.

Now, the next element whose adjacency list needs to be explored is that of t . So, I take t ; there are two vertices at adjacent to t that is u and w ; w is already been explored its color black so no need to explore it once again. So, add u to the Q set its distance from s to be 3 and color t black. And I go on like this the next vertex to be explored is x . So, the if I take x the only other vertex unexplored vertex that is incident to x is that of y , then remaining two vertices t and w are already colored black. So, I add y to the Q set its distance attribute to be 3 and color x black.

(Refer Slide Time: 19:00)

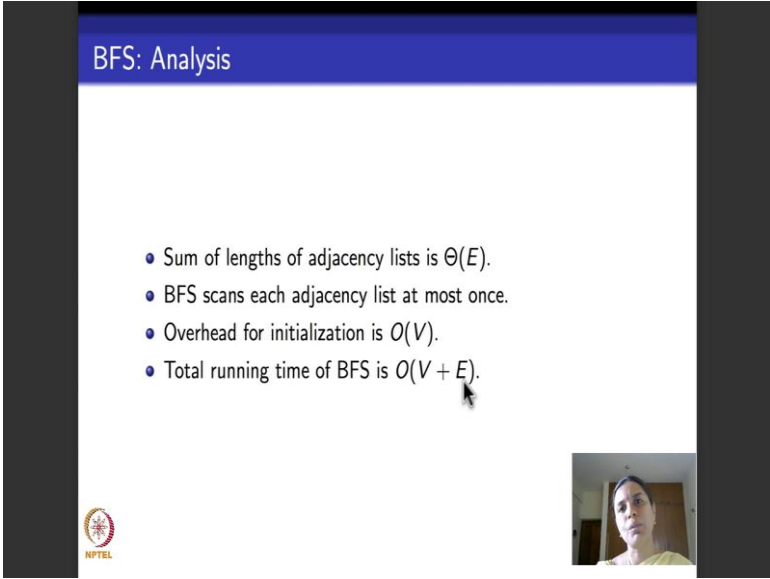


And move on like this and keep adding till I can. At some point the Q will start shrinking and the Q will become empty. So, when will the Q become empty? When I have finished all the vertices and colored all of them black.

So, if you look at this final tree how to read this tree? It says s is the source and you just read out the blue arrow edges as the tree. So, the tree has an edge from s to r and the tree goes like this s to w, w to t, w to x, x to y, t to u. So, breadth first search as an algorithm outputs this tree by saying that I have explored all the vertices and it also outputs the distance of each vertex from the tree. So, I hope the algorithm is clear to you. So, what do I do to start with I take a graph, I start with the source I explore the adjacency list of each graph that is what it means to say that I go in a breadth first way and I keep doing till I finish exploring.

And the output of breadth first search is basically a tree that I read out from the pi or the predecessor attributes, and the distance of each vertex as it occurs in the tree from the source s.

(Refer Slide Time: 20:14)



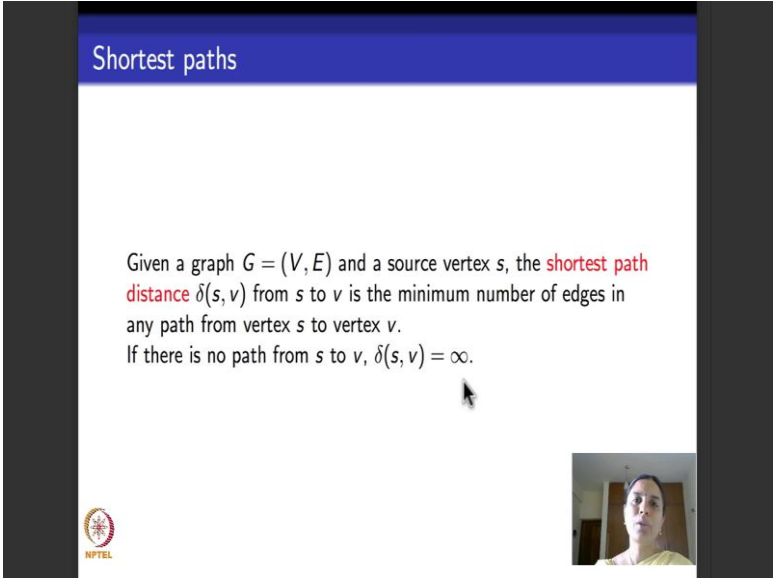
BFS: Analysis

- Sum of lengths of adjacency lists is $\Theta(E)$.
- BFS scans each adjacency list at most once.
- Overhead for initialization is $O(V)$.
- Total running time of BFS is $O(V + E)$.

So, what is the running time of breadth first search? If you go back and see the pseudo code of the breadth first search, so there is this initial for loop that runs once for each vertex, so this takes order big O of V time. And then this while loop it runs, once it enqueues a vertex it does not really go and put is back again; once it enqueues and dequeues in each vertex gets in and gets out of the Q exactly once. And for each vertex it explores for loop along the length at the adjacency list of that vertex.

So, BFS scans each adjacency list at most once and sum of the lengths of adjacency lists that we saw is theta E. And I told you the initial for loop takes big O of V. So, the total running time of BFS is big O of V plus E. So, it is a linear time algorithm that is linear in the size of the graph. So, when I talk about the size of the graph which typically consider the vertices and the edges; the number of vertices and the number of edges we do not really say only the vertices it is a wrong thing to see you take the loop size of the edges also as very much the part of the graph. So, BFS runs in time linear of the size of the graph.

(Refer Slide Time: 21:27)



Shortest paths

Given a graph $G = (V, E)$ and a source vertex s , the **shortest path distance** $\delta(s, v)$ from s to v is the minimum number of edges in any path from vertex s to vertex v .
If there is no path from s to v , $\delta(s, v) = \infty$.

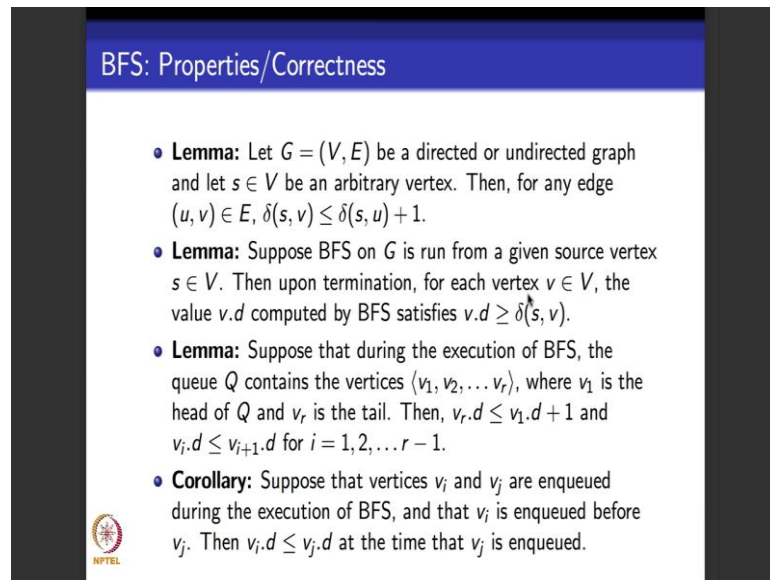
NPTEL

Video inset showing a person speaking.

So, I told you along with breadth first search it also outputs a few other things that will be useful. It outputs what is called the shortest paths distance from the source s . So, BFS runs from a fixed source and shortest path distance is the distance in the BFS tree in terms of the number of edges, it so happens that BFS outputs the shortest paths distance of each vertex from the source. Standard terminology that a book called Introduction to Algorithms by Cormen Leiserson; CLRS (Cormen Leiserson, Rivest and Stein) outputs the shortest paths distance and the notation that is used to this is delta. I have taken the pseudo code these examples of that book it is a pretty standard book.

So, it outputs the shortest path distance written as delta from the source s to each vertex V which is the shortest path in terms of the number of edges that is encountered.

(Refer Slide Time: 22:26)

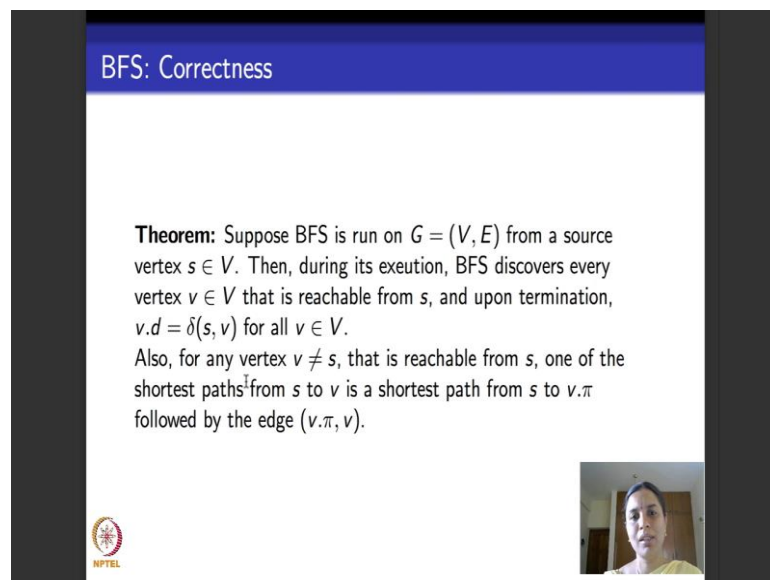


BFS: Properties/Correctness

- **Lemma:** Let $G = (V, E)$ be a directed or undirected graph and let $s \in V$ be an arbitrary vertex. Then, for any edge $(u, v) \in E$, $\delta(s, v) \leq \delta(s, u) + 1$.
- **Lemma:** Suppose BFS on G is run from a given source vertex $s \in V$. Then upon termination, for each vertex $v \in V$, the value $v.d$ computed by BFS satisfies $v.d \geq \delta(s, v)$.
- **Lemma:** Suppose that during the execution of BFS, the queue Q contains the vertices $\langle v_1, v_2, \dots, v_r \rangle$, where v_1 is the head of Q and v_r is the tail. Then, $v_r.d \leq v_1.d + 1$ and $v_i.d \leq v_{i+1}.d$ for $i = 1, 2, \dots, r - 1$.
- **Corollary:** Suppose that vertices v_i and v_j are enqueued during the execution of BFS, and that v_i is enqueued before v_j . Then $v_i.d \leq v_j.d$ at the time that v_j is enqueued.


There are several properties that you need to show that actually BFS is correct; what is it means to say BFS is correct?

(Refer Slide Time: 22:32)



BFS: Correctness

Theorem: Suppose BFS is run on $G = (V, E)$ from a source vertex $s \in V$. Then, during its execution, BFS discovers every vertex $v \in V$ that is reachable from s , and upon termination, $v.d = \delta(s, v)$ for all $v \in V$. Also, for any vertex $v \neq s$, that is reachable from s , one of the shortest paths from s to v is a shortest path from s to $v.\pi$ followed by the edge $(v.\pi, v)$.



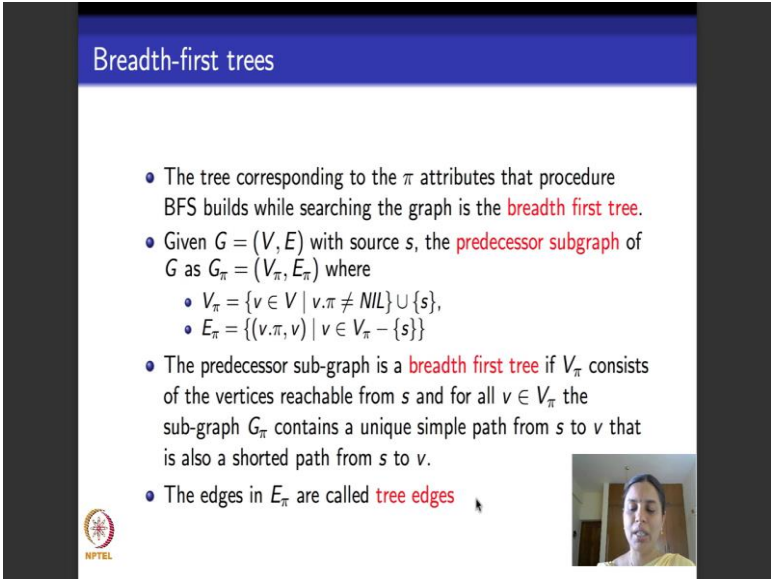
So, we will state and prove a theorem like this, let us read out this theorem. It says suppose breadth first search is run on a particular graph from a fixed source s then during its execution BFS manages to find or discover every vertex that is reachable from s in the given graph g and when it terminates it outputs the shortest path distance from s to each vertex in the graph. For each vertex that is reachable from s ; one of the shortest paths

will be the path that the breadth first tree takes. So, for vertices that are not reachable from s ; this particular example graph that we saw did not have anything like that. For vertices that are not reachable from s breadth first search will not be able to output anything because it will not explore it.

So, what you basically do is you begin breadth first search from a fresh source; that is not s then you can explore the graph once again from other source and reach all the vertices that were not reachable from the source s . And you can repeat this process to get a forest of breadth first trees till we complete. The way I have presented this pseudo code, it is presented in such a way that we present it as we are exploring from a fix source s and then we stop.

So, those vertices that are not reachable from s , if there are any in the graph will not be explored, but there is no harm in running the breadth first search algorithm again from another source to repeat the process. Please remember that. So, you can explore all the vertices in the graph instead of getting one breadth first tree you will end up getting a breadth first forest.

(Refer Slide Time: 24:11)



Breadth-first trees

- The tree corresponding to the π attributes that procedure BFS builds while searching the graph is the **breadth first tree**.
- Given $G = (V, E)$ with source s , the **predecessor subgraph** of G as $G_\pi = (V_\pi, E_\pi)$ where
 - $V_\pi = \{v \in V \mid v.\pi \neq NIL\} \cup \{s\}$,
 - $E_\pi = \{(v.\pi, v) \mid v \in V_\pi - \{s\}\}$
- The predecessor sub-graph is a **breadth first tree** if V_π consists of the vertices reachable from s and for all $v \in V_\pi$ the sub-graph G_π contains a unique simple path from s to v that is also a shortest path from s to v .
- The edges in E_π are called **tree edges**

So, now let us go ahead and discuss about how to use the π attributes to be able to get the breadth first tree. If you see this example what I have done is that a blue lined arrows are the π attributes. How do we use that to be able to output the tree? So, what do I do? I do this. Given a graph G is equal to V comma E with the source s which is basically

input to the breadth first search algorithm, I generate what is called the predecessor subgraph by using the π attributes.

So, what are the vertices of the predecessor of graph? The vertices of the predecessor of graph are all the vertices of the given graph such that they are reachable from s . Once they are reachable from s then π attribute will be non NIL right, it will not be a NIL thing because BFS will reset it to the parent vertex and then you take this source s . What are the edge set of the edges of the predecessor of the sub-graph? It will be the vertex and its predecessor, because that is how edges in the tree look like.

So, you can prove what theorem which says that such a predecessor of graph is actually a tree; in the sense that it is connected and it has no cycles. So, what breadth first search outputs is a tree or a forest of trees containing shortest path from the source to each reachable vertex. The edges of this tree are called tree edges. Here are the lemmas that we need to use to show the correctness of breadth first search, because the main focus of this course is not on algorithms, I have just stated the lemmas and not proved their correctness. Feel free to refer to a book like CLRS to check for correctness or proofs of all these lemmas.

So, initially you need a lemma which says that the shortest path distances increases as I explore the graph. So, it says when we start from the source then you reach a vertex u from the source having $d[s, u]$ has now to be, and suppose u, v is an edge in the graph then what is the shortest path distance from s to v . It is at most the shortest path distance from s to u plus 1. Because I need to be able to consider the paths to reach from s to u and then consider the edge u, v , it cannot be more than that this is popularly called triangular inequality.

The second lemma says that suppose you run BFS on a graph from the given source vertex then when this algorithm terminates, for each vertex v , the value the distance $d[v]$ the BFS algorithm computes will be at most at least the shortest path distance; sorry it will be at least the shortest path distance, it could be more in fact it will be equal to the shortest path distance for this kind of breadth first search algorithm. Third lemma says that as I go on exploring the graph, the shortest path distances increases. It monotonically increases that is what the third lemma says.

Fourth lemma says that the Q respects the order in which, the Q strongly influences the order of exploring the vertices. For example, if you go back and see in this code, right in the beginning, when I look at the source s right it has two vertices in adjacency list r and w; I put w first and r next. And I told you at that itself that there is no sanctity about it I could put r first and w next. I would have got a slightly different breadth first tree, but nonetheless there would still be the shortest path distance from the source. That is what this lemma says. It says that the queue Q respects the distance enqueue and dequeue respects the distances of the vertices.

So, using these lemmas, I will be able to show correctness in BFS. Correctness in BFS basically says that I do BFS transverse it will finish exploring every vertex that is reachable from the source, and it will output the shortest path distance in terms of the number of matrix from that source to every other vertex. So, this is all I had to tell you about BFS. In the next module we will look at depth first search and also we will look at algorithms for connected components in the graph.

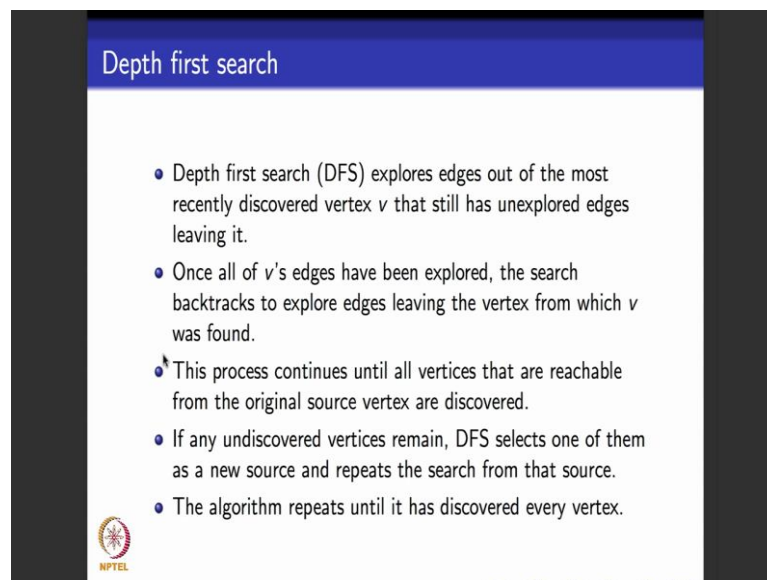
Thank you.

Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 08
Elementary Graph Algorithms

Hello again. The goal of today's lecture is to be able to do depth first search algorithm which is another popular graph algorithm followed by strongly connected components how to use depth first search to output what are called strongly connected components in a graph.

(Refer Slide Time: 00:30)



Depth first search

- Depth first search (DFS) explores edges out of the most recently discovered vertex v that still has unexplored edges leaving it.
- Once all of v 's edges have been explored, the search backtracks to explore edges leaving the vertex from which v was found.
- This process continues until all vertices that are reachable from the original source vertex are discovered.
- If any undiscovered vertices remain, DFS selects one of them as a new source and repeats the search from that source.
- The algorithm repeats until it has discovered every vertex.

NPTEL

So, we will begin with depth first search. Unlike breadth first search that we saw in the last module, depth first search is also meant to traverse or explore the vertices of the graph, but in a depth first way. So, it goes deep down a graph as much as possible. It begins at a particular designated source like BFS and then instead of exploring the adjacency list of that source fully it takes one successor from that source goes to that successor, then it picks up one successor of the successor goes to that successor and so on. So, it goes deeper in the graph and first finishes exploring the graph to the deepest possible path that it can trace from the source.

Then it backtracks comes back and picks up the next vertex in the adjacency list of the source, and goes deep down that vertex. And when it finishes going down for that vertex

It comes back, picks up the next adjacency vertex adjacent to s , goes deep down and when it finishes exploring the adjacency list of each vertex deep down it finally, comes back and colors s black right. So, the goal of depth first search is also to traverse a graph and produce a tree, but unlike depth first search it goes deeper down in the traversal. The breadth first search grows breadth first in the traversal.

(Refer Slide Time: 01:57)

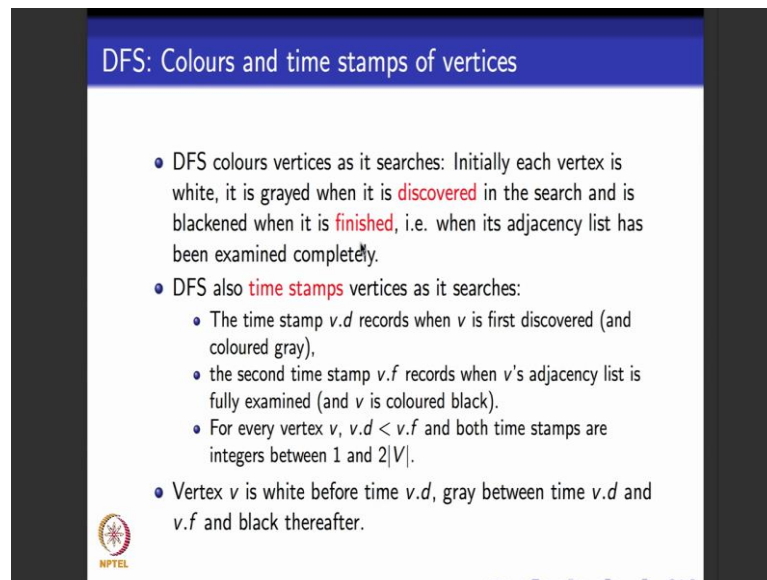
Depth first forest

- The predecessor sub-graph of G is given by $G_\pi = (V, E_\pi)$ where

$$E_\pi = \{(v, \pi, v) \mid v \in V \text{ and } v.\pi \neq NIL\}$$
- The predecessor sub-graph of DFS forms a depth-first forest comprising several depth first trees.
- The edges in E_π are called tree edges.

Like BFS, DFS also keeps several attributes it keeps the π predecessor attribute associated with each vertex, it keeps a color associated with each vertex it keeps 2 kinds of time stamps unlike breadth first search I will tell you what they are very soon, but what does the π attribute look like? π attribute basically is useful to produce or output the depth first tree right, set of depth first tree from different sources we will constitute depth first forest and the edges in this tree are called tree edges. So, as and when I explore the graph deep wise, I said the π attribute of each vertex that I encounter to be it is parent, and when I consider all the π attributes this way I get the full predecessor sub graph which happens to be a tree or a forest of trees based on whether the graph has one connected component reachable from the source several connected components reachable from the source.

(Refer Slide Time: 02:54)



DFS: Colours and time stamps of vertices

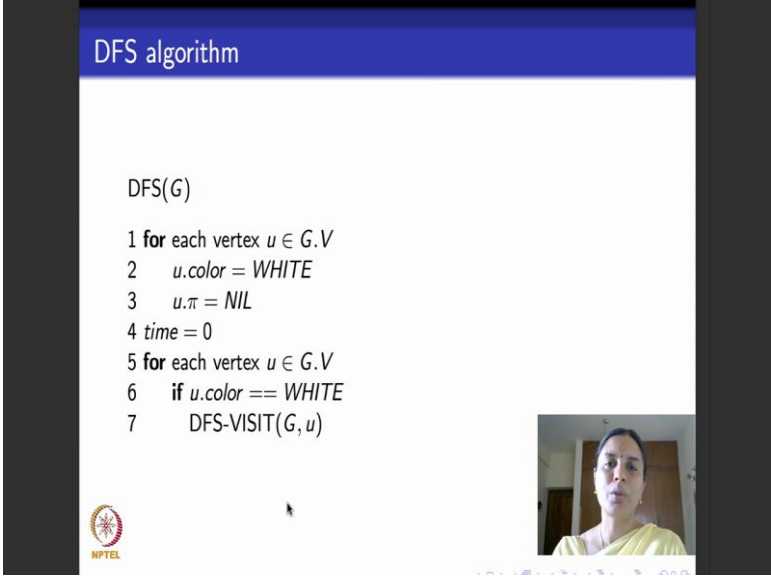
- DFS colours vertices as it searches: Initially each vertex is white, it is grayed when it is **discovered** in the search and is blackened when it is **finished**, i.e. when its adjacency list has been examined completely.
- DFS also **time stamps** vertices as it searches:
 - The time stamp $v.d$ records when v is first discovered (and coloured gray),
 - the second time stamp $v.f$ records when v 's adjacency list is fully examined (and v is coloured black).
 - For every vertex v , $v.d < v.f$ and both time stamps are integers between 1 and $2|V|$.
- Vertex v is white before time $v.d$, gray between time $v.d$ and $v.f$ and black thereafter.

NPTEL

So, like BFS, DFS also goes ahead at first discovers a vertex and then it also finishes the vertex. Now what we do is unlike DFS we keep 2 different kinds of time stamps here, a time stamp called d which is given when a vertex is first discovered and which means a vertex which was originally colored white now becomes colored grey. And then another time stamp called f , f for finish time stamp which is given when the vertex is colored black right. When the adjacency list that vertex is fully examined. So obviously, a vertex first needs to be discovered before its adjacency lists is fully explored at the vertex is finished.

So, the d timestamp that is given to a vertex is always strictly less than the f timestamp that is given to the vertex. And the d and the f timestamps cannot be more than the number of vertices in the graphs, because that that many paths could be there assuming that the whole graph is connected one for d timestamp and one for f timestamp. So, they are basically values between 1 and $2 \bmod V$. So, before it is given the d timestamp the color of a vertex is white. Between when it is given the d timestamp and the f timestamp, it is color is grey and when it is colored black we give the finish timestamp to a vertex. So, here is how the algorithm looks like. I have split this algorithm across 2 slides because I could not fit in to one slide.

(Refer Slide Time: 04:24)



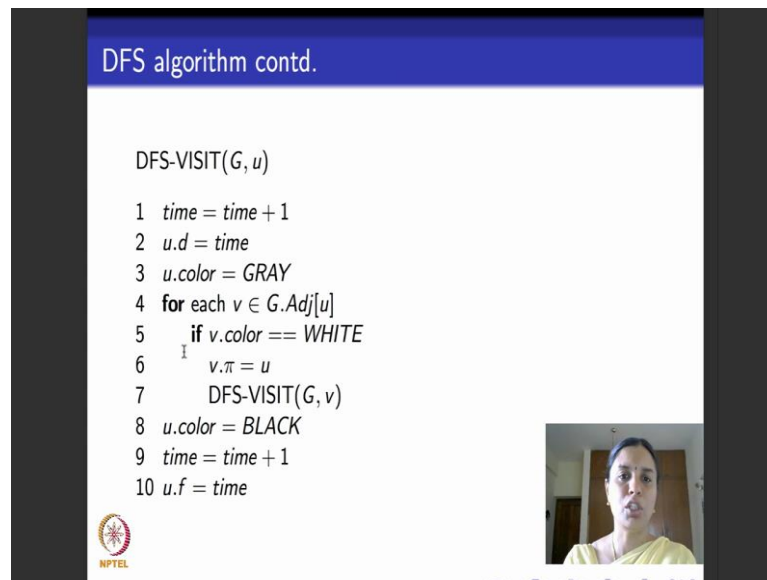
The slide is titled "DFS algorithm" in a blue header. Below the header, the text "DFS(G)" is followed by a list of seven steps:

- 1 for each vertex $u \in G.V$
- 2 $u.color = WHITE$
- 3 $u.\pi = NIL$
- 4 $time = 0$
- 5 for each vertex $u \in G.V$
- 6 if $u.color == WHITE$
- 7 $DFS-VISIT(G, u)$

In the bottom left corner, there is a small circular logo with the text "NPTEL" below it. In the bottom right corner, there is a small video inset showing a woman with dark hair wearing a yellow sari, speaking.

So, this is the initialization part as done for breadth first search. You start for every vertex you color at white you set it is pi attribute to nil. If you remember in the BFS code, we had set it is distance attribute from s to 0. Here we don't do the distance attribute we do the discovery and finish time stamps. For that I need a generic variable called time which I will use to set the dot d and the dot f timestamp. So, I initialize the generic variable time here to be 0. So, after I have done this, what I do is for every white colored vertex in the graph I begin this procedure called DFS visit from that vertex. What does the procedure DFS visit to? DFS visit first thing it does is increments the timestamp because it is beginning to search from a new vertex and it says this new vertex from where I am beginning my search which is the vertex u is discovered now.

(Refer Slide Time: 05:17)



DFS algorithm contd.

```
DFS-VISIT( $G, u$ )  
1   $time = time + 1$   
2   $u.d = time$   
3   $u.color = GRAY$   
4  for each  $v \in G.Adj[u]$   
5      if  $v.color == WHITE$   
6           $v.\pi = u$   
7          DFS-VISIT( $G, v$ )  
8   $u.color = BLACK$   
9   $time = time + 1$   
10  $u.f = time$ 
```

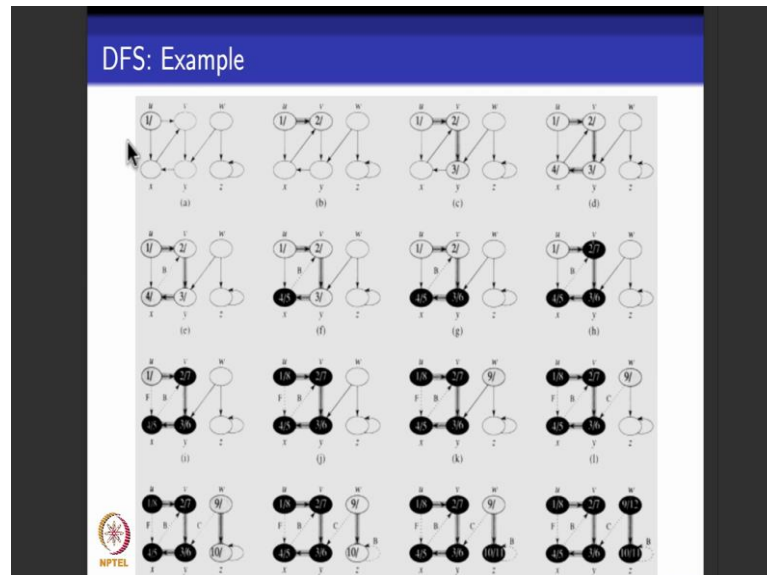
NPTEL

So, it sets the timestamp that it is just incremented as the discovery timestamp for the vertex u , and because I am going to begin exploring from the vertex u which was white, I set the color attribute of u to be grey. Now I start exploring the adjacency list of u . So, if there is a white colored vertex in the adjacency list u , I have to color that vertex grey and then start exploring deep down that vertex v . So, I pick up the white colored vertex call it v that is present in the adjacency list of u , and then the first thing that I do is I say u is the predecessor of v in my depth first tree, because I found v as I was exploring u . Now I have to go down and deeply explore the path that come out of v right. So, at v I recursively call the same procedure DFS visit. So, assuming that this code gets instantiated here, what would you do, you increment the timestamp you say that the vertex v is discovered, set its timestamp, set color of v to grey and go inside the adjacency list of v .

If you find another white vertex then you call the same procedure again for that vertex in the adjacency list. So, this way DFS let us you go down the path corresponding to a particular vertex. And when you finish exploring all the paths deep down from a particular vertex u , which means you finished exploring this recursive call when you come out of this recursive call for every vertex in the adjacency list of u , then you say I have finished exploring the adjacency list of u fully. Then you color u black, increment the time right, time variable that you kept as a global variable, at say that u is finished

because it is colored black and set is finished timestamp to be the current value of time right.

(Refer Slide Time: 07:39)



So, we will see an example to make it clear. What I have done this I have squeezed in the graph all in one slide. I hope you can read this which is not what I did for breadth for search here I have squeezed the whole thing into one slide to make it better understandable.

So, here is this graph you look at the top left corner that is the given graph. How many vertices does it have? It has 6 vertices u, v, w, x, y and z right. Unlike breadth for search here just for illustrative purposes, I have followed CLRS and taken an example which is a directed graph. This is the example from the same book by CLRS. So, what I do is u is the source from where I begin my depth first search. Now if you look at this graph a bit before you start doing depth first search of the graph, you realize that the vertices v, x and y are reachable from u, but the vertices w and z are not reachable from u. If you notice there is no path, no edge that connects either of these vertices to w and to z. So, when I explore starting from u, I will be able to explore only these 4 vertices. And then I have to start my DFS search again fresh from w to be able to explore the remaining two vertices w and z. And what will be the output of the algorithm? It will be a forest containing 2 trees. One with u as the root or the source of exploration, and one with w as the source of the root of the next depth first tree.

So, will begin with u. So, what is this label inside u? Read it has one slash nothing. So, there are 2 parts to a label there is a numerator apart thing of this label as a number having like a fraction having a numerator and the denominator. The numerator part, the part on the top, the left side of the fraction indicates the discovery time the dot d time of a vertex. The denominator part indicates the finish time or the dot f time the vertex. Right now to begin with because I am exploring DFS from u as the source vertex, I say discovery time of u is one, because I am exploring. Not yet finished exploring the adjacency list of u fully. So, no finish time is assigned to u. So, the right hand side of the bottom part which contains the finish time is left black. Now I explore the adjacency list of u as per what this pseudo call says. How many vertices are there if you see in this graph? There is a v in the adjacency list of u there is x in the adjacency list of u. I can choose either of them this particular example let say be go to v.

So, v which was originally colored white now becomes grey color. Again I have not depicted like in BFS, I have not depicted the color of the vertices here, the grey color is not when depicted just the black color is when depicted for clarity sake. So, I have discovered v, my timestamp is got incremented to two. So, I have sign discovery timestamp of v to 2 and then I say this shade of this arrow indicates that in the depth first tree u is the predecessor of v. The pi attribute assigned to v assigns the value as u. Now what do I do? I have to recursively call the procedure DFS visit from the vertex v which means I have to explore the adjacency list of the vertex v. In this example it just so happens that there is only one vertex adjacent to v in the adjacency list which happens to be y.

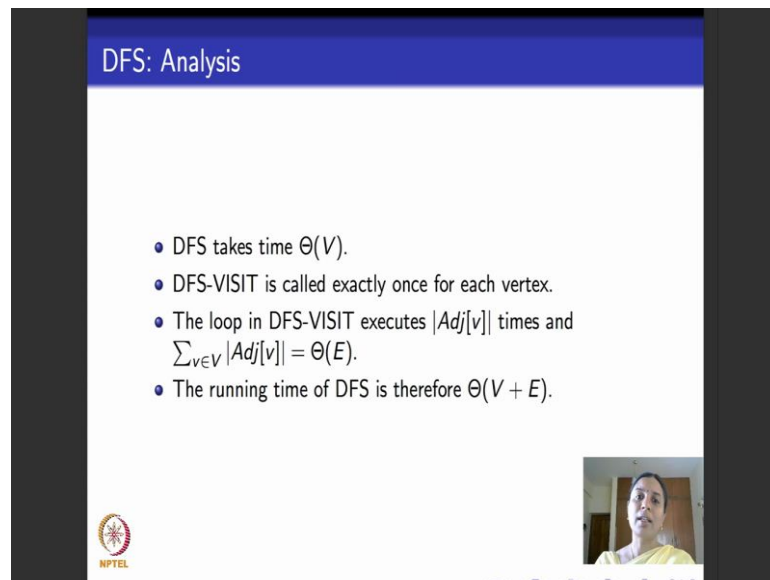
So, I say I go their y has been discovered I assign it is discovery timestamp to be 3, and then I said the predecessor y to be v in the DFS tree. Now I start exploring recursively the adjacency list of y. If you notice, I am going deeper in the graph right. I did not bother after doing v to come to x, because I am not doing breadth first search. After doing v I went to it is successor y because I am doing depth first search. Now I explore the adjacency list of y. What is there in the adjacency list of y in this example? It is just x. So, I discover x, assign it is discovery timestamp to be 4, and set y as the predecessor of x by putting the shade. So, so far the depth first tree that I have generated looks like this. It has these 4 vertices u, v, y and x and the edges of the tree are these grey shadow edges right. Now I repeat the procedure, I have to look at the adjacency list of x. If you

look at the adjacency list of x what is the only vertex that is present in the adjacency list of x, that is v.

But remember v is already being discovered. So, v is not colored white right. So, I do not go back and read discover v right. So, I let it be there is no other vertex of the adjacency list of this vertex x. So, which means I have finished my exploring my vertex, my search from the vertex x. So, I assign a finish stamp of 5 which is one more than the discovery stamp of 4, and then I color x black. I move on, I repeat this and then I say I have come back to the same thing whatever was holding for x holds for y, there is no more to explore. So, I finish y color it black, assign a finished time. And then I move back to v same thing I finish v, color it black assigned a finished time move back to u, finish u, color it black assigned a finish time. No more to explore at this stage if you look at this graph that is labeled with j, I finished exploring the full thing, the remaining 2 vertices were not reachable from my source u. Those 2 vertices were w and x.

So, I start fresh ones more depth first search from the vertex w. The last finish timestamp I had was 8. So, assigned for discovery timestamp of 9 to w, start and repeat the same exploration from w. In this case I finish first enough through all these edges because there are only 2 vertices that nothing more to it, and this path again leads to all already black colored vertex. So, I repeat the same procedure for w and get this. So, the final output of my DFS algorithm we will 2 depth first trees. One that has its root at u and has all these vertices, the grey colored entity is the tree that you draw out, and the next depth first tree in the forest has it is root as w and I just has these 2 vertices and one edge. So, this is how DFS works.

(Refer Slide Time: 14:22)



DFS: Analysis

- DFS takes time $\Theta(V)$.
- DFS-VISIT is called exactly once for each vertex.
- The loop in DFS-VISIT executes $|Adj[v]|$ times and $\sum_{v \in V} |Adj[v]| = \Theta(E)$.
- The running time of DFS is therefore $\Theta(V + E)$.

NPTEL

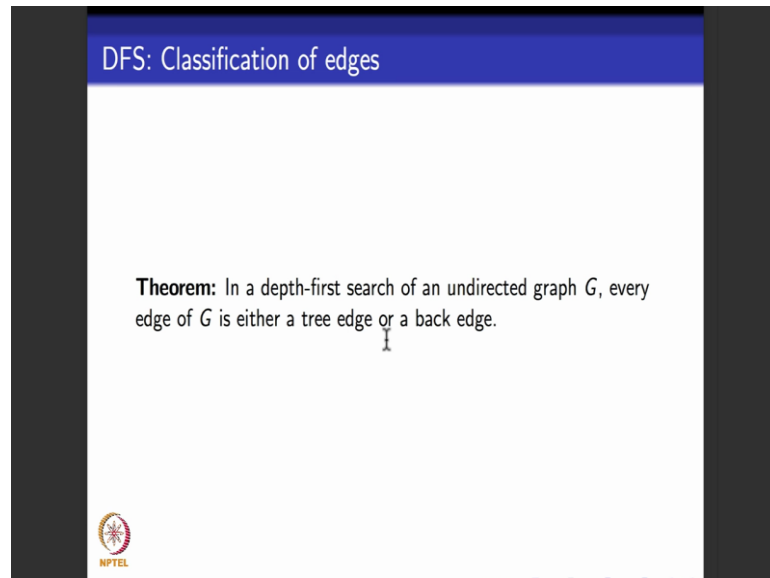
What is the running time of DFS? Let us go back and look at the code this for loop which does the initialization takes mod v time, and then the procedure DFS visit, how many times does it run? It runs at most once for every vertex that is reachable through an edge in the graph from the source. So, the initial thing takes order V time. DFS visit is called exactly once for each vertex it does not call a vertex that is already colored black.

So, the loop in DFS visit if you see there is one more loop here this loop in this recursive procedure DFS visit, runs at most this time right, at most the cardinality of the adjacency list of a particular vertex. Sum of all the adjacency list is no more than the number of adjust as we saw. So, the total running time of depth first search is also V plus E . So, depth first search also runs in time linear in the size of the graph. So, pretty much it is the same as breadth first search they have no difference except in terms of the convenience of what you want to use. I would ideally say that if you are not sure about what is the kind of graph it is you go for breadth first search because it is a safer way to explore. If we go for depth for search you might entire a loop in a graph that corresponds to the control flow graph and the loop could be infinite and you may not be able to come out of the loop.

So, breadth first search is slightly better to explore a graph when you are not sure about the kind of graph that you are looking at. So, towards showing correctness of depth first search again I will not be able to spend time proving the correctness of this algorithm.

What I will do is walk you through the results that finally, show that depth first search is correct.

(Refer Slide Time: 16:02)



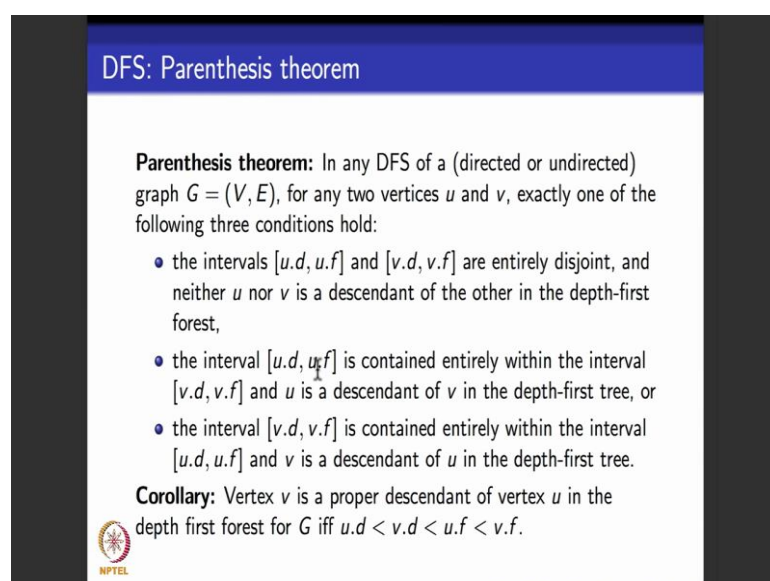
DFS: Classification of edges

Theorem: In a depth-first search of an undirected graph G , every edge of G is either a tree edge or a back edge.

NPTTEL

So, what is the main correctness result that I want to show about depth first search, that is this. So, when I take a graph and run depth first search on the graph then, depth first search explores the graph and it produces a forest of depth first trees containing tree edges or back edges.

(Refer Slide Time: 16:22)



DFS: Parenthesis theorem

Parenthesis theorem: In any DFS of a (directed or undirected) graph $G = (V, E)$, for any two vertices u and v , exactly one of the following three conditions hold:

- the intervals $[u.d, u.f]$ and $[v.d, v.f]$ are entirely disjoint, and neither u nor v is a descendant of the other in the depth-first forest,
- the interval $[u.d, u.f]$ is contained entirely within the interval $[v.d, v.f]$ and u is a descendant of v in the depth-first tree, or
- the interval $[v.d, v.f]$ is contained entirely within the interval $[u.d, u.f]$ and v is a descendant of u in the depth-first tree.

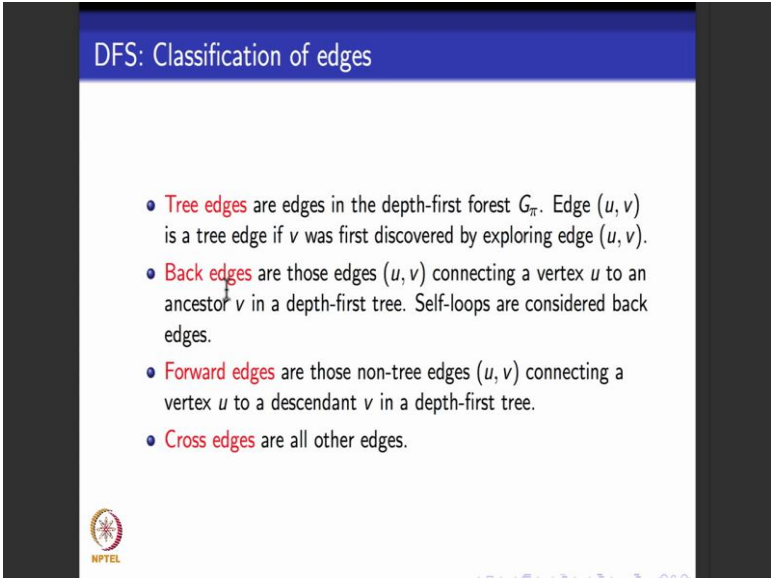
Corollary: Vertex v is a proper descendant of vertex u in the depth first forest for G iff $u.d < v.d < u.f < v.f$.

NPTTEL

So, towards that I have all these theorems. So, this parenthesis theorem basically says that the discovery and finish times of all the vertices are in proper intervals. If you see this example right I continuously increase, I first discovered this time is one I next discover this timestamp is 2 then I next discovered this time stamp is 3, and when I finish your timestamp is 5, right. And if you see if I go back and finish the vertex u , its timestamp is 8 which is greater than the timestamp 5.

So, this lemma says that v is a proper descendant of a vertex u in the depth first forest then the timestamp that is assigned, the discovery, u assigned discovery timestamp first and then it is successor descendant v is discovered - u is finished and then v is finished right.

(Refer Slide Time: 17:17)



DFS: Classification of edges

- **Tree edges** are edges in the depth-first forest G_π . Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v) .
- **Back edges** are those edges (u, v) connecting a vertex u to an ancestor v in a depth-first tree. Self-loops are considered back edges.
- **Forward edges** are those non-tree edges (u, v) connecting a vertex u to a descendant v in a depth-first tree.
- **Cross edges** are all other edges.

NPTTEL

So, there are 4 kinds of edges that depth first search returns. One is what is called tree edges, the edges that belong to one of the depth first trees. The next kind is what is called back, edges that connect back, that connect a descendant back to its ancestor what are called back edges. Forward edges are edges that follow the same direction as that of tree edges, but they sort of cut across several descendants and directly connect an ancestor to a descendant.

All other kinds of edges what are called cross edges. If you go back and take this example this is the final output, right, look at the last graph that my cursor is in the final depth first tree is, looking at it here. So, that edges that colored grey what are called tree

edges. This edge from u to x marked f is a forward edge because it connects u to one of its descendants x . This edge from x to v marked b is what is called a back edge because it connects a descendant back to one of its ancestors. Similarly, this self-loop is also a back edge and this kind of an edge which is not a forward edge, not a tree edge, not a back edge is what is called a cross edge right. There should be a first categorizes edges into 4 parts.

(Refer Slide Time: 18:51)

Some definitions

Given a graph $G = (V, E)$,

- A **strongly connected component** of G is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v in C , u and v are reachable from each other.
- The **transpose** of G is given by $G^T = (V, E^T)$ where $E^T = \{(v, u) \mid (u, v) \in E\}$.
- G and G^T have exactly the same strongly connected components: u and v are reachable from each other in G iff they are reachable from each other in G^T .

The slide includes the NPTEL logo in the bottom left and a small video inset of a person in the bottom right.

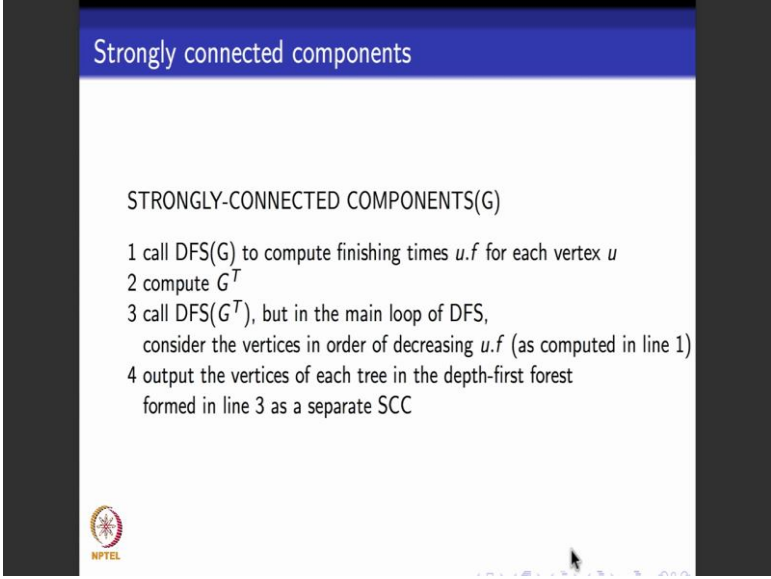
Now, the other thing that I wanted to do is to tell you how to use DFS to output what are called strongly connected components in a graph.

So, you need a directed graph to have strongly connected components and if you have an undirected graph you output what are called connected components. So, what is strongly connected component? A strongly connected component is a sort of a cycle in a directed graph. So, it says a subset of vertices is a strongly connected component if for every pair of vertices u, v in that component u is reachable from v and v is reachable from u . So, if you go back and look at this graph that we had in the slide here, if you see this graph, this is a strongly connected component, v, y, x . They, all 3 of them are reachable from each other through this cycle that I'm tracing out now, through the cursor. Similarly, just this is z is another strongly connected component, single vertex strongly connected component. If you see w cannot belong to this strongly connected component because w can be reached from y , but not vice versa.

So, this graph has two strongly connected components one that has this triangle and one that has this self-loop. So, I want to be able to know how to use DFS to be able to output the strongly connected components in the graph because I will use them to be able to look at prime paths in other entities for test case generation. So, to do that I look at the graph and I also look at its transpose. What is a transpose of a graph? You take the same graph and you reverse the directions of the edges assuming that it is a directed graph? So, a transpose of this graph in this example would be the same graph in terms of the vertices, but every edge will be presented with its direction reversed. So, I look at the transpose of the graph. One thing to note is that G , the graph and its transpose have the same strongly connected components right. Because if one vertex was reachable from the other in the original graph then the same property would hold in the reversed graph. So, I just go in the reverse way.

They were reachable from each other in both directions right. So, to be able to output the strongly connected components, a standard technique to do is to be able to run DFS.

(Refer Slide Time: 21:02)



Strongly connected components

STRONGLY-CONNECTED COMPONENTS(G)

- 1 call $\text{DFS}(G)$ to compute finishing times $u.f$ for each vertex u
- 2 compute G^T
- 3 call $\text{DFS}(G^T)$, but in the main loop of DFS, consider the vertices in order of decreasing $u.f$ (as computed in line 1)
- 4 output the vertices of each tree in the depth-first forest formed in line 3 as a separate SCC

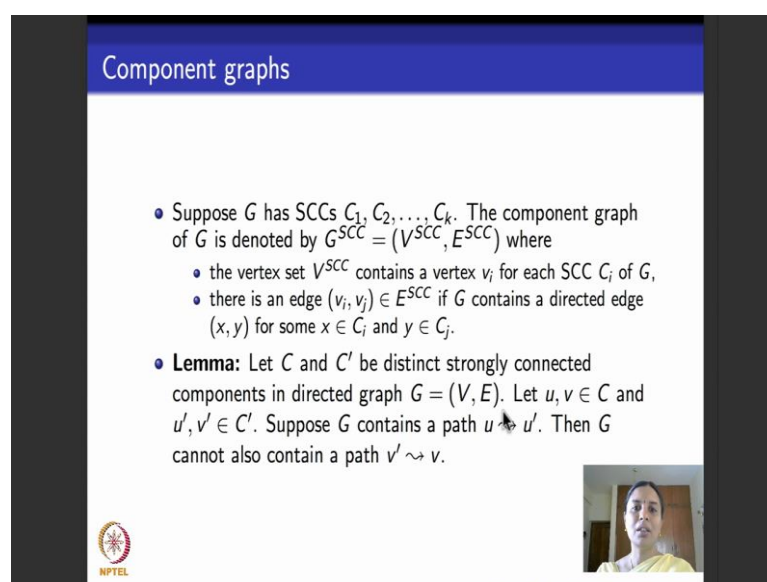
NPTL

once on the graph take its transpose and run DFS once on the transpose, but in the reverse direction of the finish times right. So, that is what this pseudo code does. You first run DFS compute finish times for each vertex u right. So, that you give you a forest of DFS trees. Then you compute the transpose of the given graph. Now you run DFS on

the transpose, but in the main loop of the DFS algorithm you consider vertices in the order of decreasing finished time. So, what it is intuitively saying is if you go back to this example after running DFS on this graph I get something like this right. Now what I do is I take the same graph take its transpose. So, I reverse the direction of every edge.

So, I get the sort of a reversed graph. So, I run DFS once again, but in the reverse direction from the highest finished time vertex. So, in some sense this tree would have traced this, if I take this strongly connected component, what I am trying to do is I am trying to traverse one half of the strongly connected component through one DFS and I am trying to traverse the other part of the strongly connected component by running DFS once again on the transpose. That is what this code is trying to do. So, how do you find strongly connected components? You run DFS on the given graph, record the finished times, take the transpose of the graph, run DFS again on the transpose graph, but in decreasing order of the finish times that you recorded in step number one. And then what you do is that you output each vertex, because you would have gone through it once this way once that way both the ways is the strongly connected components. So, each DFS tree in the DFS forest would be a separate strongly connected component that you can output right.

(Refer Slide Time: 22:51)



Component graphs

- Suppose G has SCCs C_1, C_2, \dots, C_k . The component graph of G is denoted by $G^{SCC} = (V^{SCC}, E^{SCC})$ where
 - the vertex set V^{SCC} contains a vertex v_i for each SCC C_i of G ,
 - there is an edge $(v_i, v_j) \in E^{SCC}$ if G contains a directed edge (x, y) for some $x \in C_i$ and $y \in C_j$.
- Lemma:** Let C and C' be distinct strongly connected components in directed graph $G = (V, E)$. Let $u, v \in C$ and $u', v' \in C'$. Suppose G contains a path $u \rightsquigarrow u'$. Then G cannot also contain a path $v' \rightsquigarrow v$.

So, these are lemmas that tell you that the algorithm for running DFS is basically correct. So, what they tell you is that you take the given graph, take all its strongly connected



components. Lets say it has k is strongly connected components. You collapse each strongly connected component to create a meta vertex right. So, in this example if I see, I told you right, this is one strongly connected component, this is one strongly connected component, these 2 standalone separate strongly connected components, single vertex they want have any significant, but they are like that. So, what I do is I take this and I collapse and create one vertex with this, one vertex for this entire strongly connected component, one vertex for this and one vertex for this. So, I have 4 vertices. These edges get absorbed, these edges that connect these meta connected strongly connected components get retained the component graph.

So, that is how I create the component graph. So, I say vertex is are the set of vertices for each strongly connected components there is one meta vertex. And I say if that is an edge that connects one strongly connected component to the other, then you put an edge in the new graph right. This one says that if there are 2 distinct strongly connected components in the given graph, and I take a vertex path from one strongly connected component in the meta graph to the other strongly connected component in meta graph then, they cannot be a path in the reverse direction. Basically what it says is this component graph corresponding to a given graph where I collapse this strongly connected components into a single vertex is an acyclic graph.

(Refer Slide Time: 24:37)

Discovery and finish times: Sets of vertices

- Discovery and finish times refer to those computed in the first call of DFS in the algorithm for SCCs.
- If $U \subseteq V$, $d(U) = \min_{u \in U} \{u.d\}$ and $f(U) = \max_{u \in U} \{u.f\}$.
- **Lemma:** Let C and C' be distinct SCCs in directed graph $G = (V, E)$. Suppose that there is an edge $(u, v) \in E$, where $u \in C$ and $v \in C'$. Then $f(C) > f(C')$.
- **Corollary:** Let C and C' be distinct SCCs in directed graph $G = (V, E)$. Suppose that there is an edge $(u, v) \in E^T$, where $u \in C$ and $v \in C'$. Then $f(C) < f(C')$.

So, using the fact that it is a cyclic graph now what I relate is I relate the discovery and finish times right. So, what I do is in this particular lemma one thing to be noted is that when I talk about discovery and finish times, I discuss I talk about the times that were recorded by the first DFS that runs in this algorithm, where the first DFS not by the second right. So, for an entire component I says discovery time is the least of discovery times of all the vertices. For an entire component, finish time is the highest of the finish times of all the vertices. So, what I do is suppose I have distinct strongly connected component C and C' , then if there is an edge in the direction u to v where u is in C and v is in C' then, I say that C would have been finished C' would have been finished before C right.

So, corollary is the reverse. So, it says that if c and c' are distinct strongly connected components in a given graph, and suppose there is an edge from u to v in the transpose graph then the reverse of that holds right. So, what it basically says is that if I run DFS once here and if I run DFS once in the transpose, but taking it in the decreasing order of finished times, then I will be able to distinctly identify strongly connected components individually in this graph. Why does that hold true that holds true? Because if I compose the meta graph considered the meta graph where I compose and collapse each strongly connected component into one vertex then a meta graph is acyclic, so if I run DFS on that acyclic graph I would have correctly done both the DFS right. So, the algorithms were strongly connected components that we saw here is basically correct.

So, to summarize what I wanted to recap through these 2 modules was to teach you basic graph there was an algorithms depth first search breadth first search and they have several different applications strongly connected components is one application. Similarly, you can do topological sort elementary graph such algorithms where you keep extra parameters extra tax can all be done using basic manipulations of breadth first search and depth first search, and both these algorithms have linear running time. What we will do in the next module is to see how to use this algorithm to be able to define algorithms for test requirements and test path generation to satisfy the test requirements.

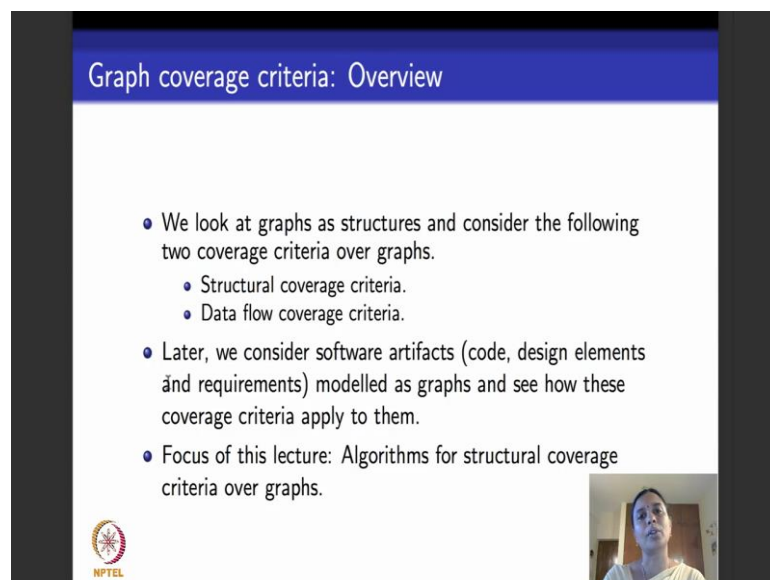
Thank you.

Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 09
Algorithms: Structural Graph Coverage Criteria

Hello everyone. In this module our main focus would be to get backs to structural coverage criteria on graphs that we saw two modules ago, and look at algorithms that will help us to write down the test requirements for each of the coverage criteria we saw. And also look at algorithms that will help us to generate test paths that will meet the test requirements for these algorithms.

(Refer Slide Time: 00:37)



The slide is titled "Graph coverage criteria: Overview" in a blue header. It contains three bullet points:

- We look at graphs as structures and consider the following two coverage criteria over graphs.
 - Structural coverage criteria.
 - Data flow coverage criteria.
- Later, we consider software artifacts (code, design elements and requirements) modelled as graphs and see how these coverage criteria apply to them.
- Focus of this lecture: Algorithms for structural coverage criteria over graphs.

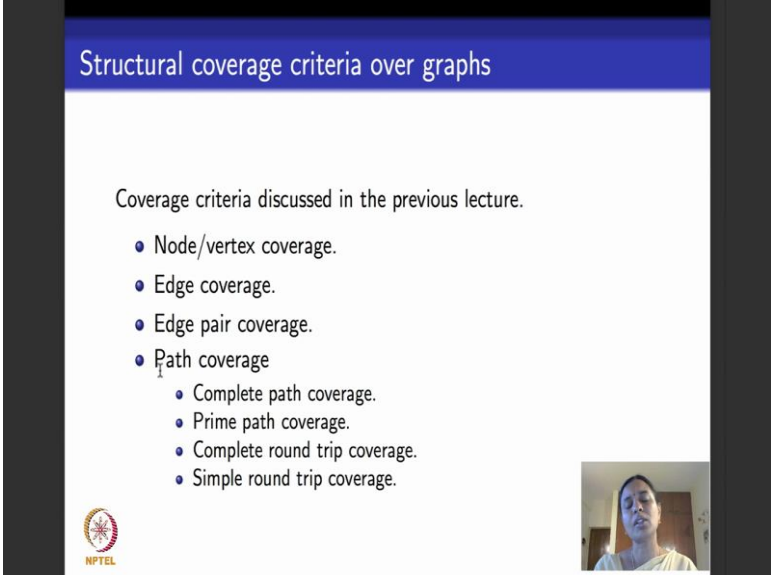
In the bottom left corner, there is a small circular logo with the text "IITEL" below it. In the bottom right corner, there is a small video inset showing a person speaking.

So, just to recap the overall module that we are looking at currently. We are now looking at designing test cases where software artifacts are modeled as graphs. So, in the graph models we consider two kinds of coverage criteria: coverage criteria based on the structures of the graph which we saw two modules ago. And in the next week we will look at coverage criteria on graphs augmented with variables, data talking about variables and see data flow coverage criteria.

Moving on from there we will see how various software artifacts can be modeled as these graphs, and how these coverage criteria can be used for designing test cases. Now

what will be the focus of this lecture? We look at algorithms for structural coverage criteria in this lecture.

(Refer Slide Time: 01:24)



The slide is titled "Structural coverage criteria over graphs" in a blue header. Below the title, it says "Coverage criteria discussed in the previous lecture." followed by a bulleted list:

- Node/vertex coverage.
- Edge coverage.
- Edge pair coverage.
- Path coverage
 - Complete path coverage.
 - Prime path coverage.
 - Complete round trip coverage.
 - Simple round trip coverage.

In the bottom left corner, there is an NPTEL logo. In the bottom right corner, there is a small video inset showing a person speaking.

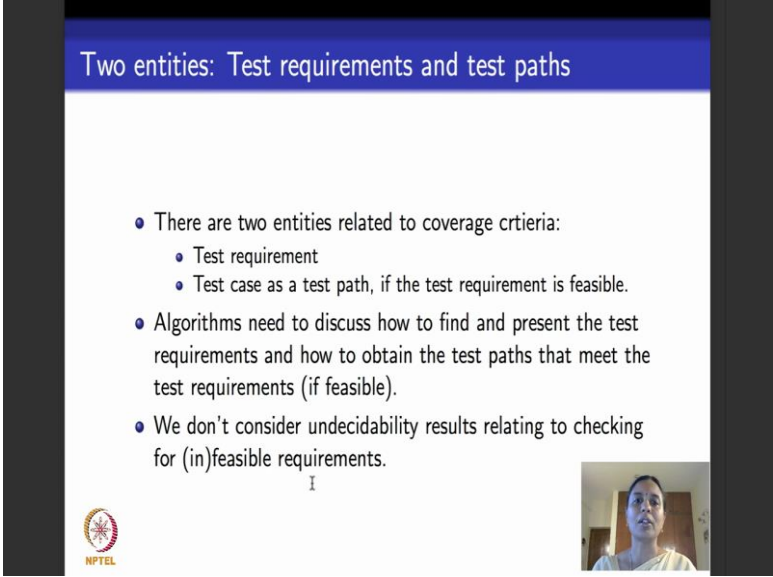
So, to recap what was the structural coverage criteria over graphs that we saw about two lectures ago. If you remember we saw all these coverage criteria we began with node or vertex coverage, then we moved on to edge coverage, we looked at edge-pair coverage, and then we looked at path coverage in graphs. We began with complete path coverage which I told you was infeasible if a graph has a single loop or a self loop, because you can go round and round the loop several times and get infinite number of paths. So, there is no notion of the finite path set that I can completely cover.

So, a work around would be to do specified path coverage. Specified path coverage is where user gives the paths. Another interesting path coverage criteria that has existed in the testing literature is that are prime paths coverage. These specially help to cover graphs with loops by helping us to execute the loop once, execute the loop many times, and also to skip the loop. And then prime paths that begin and end with the same node what are called round trip coverage.

So, we ended the structural coverage criteria lecture by looking at two round trip coverage criteria. One was complete round trip coverage, which insisted that you cover all the round trips. The next one was simple round trip coverage which insisted that you cover one of the round trips.

So, what we will be looking at now is, fine, we have these coverage criteria each coverage criteria has its set of test requirements as per the criteria that is under consideration, and then our goal as a tester is to be able to define test paths for those coverage criteria. How does one go about doing it? So, what are the algorithms that will help us to do this?

(Refer Slide Time: 03:14)



The slide has a blue header with the title "Two entities: Test requirements and test paths". Below the header is a white area containing a bulleted list. In the bottom right corner of the slide, there is a small video inset showing a person speaking. The NPTEL logo is in the bottom left corner.

- There are two entities related to coverage criteria:
 - Test requirement
 - Test case as a test path, if the test requirement is feasible.
- Algorithms need to discuss how to find and present the test requirements and how to obtain the test paths that meet the test requirements (if feasible).
- We don't consider undecidability results relating to checking for (in)feasible requirements.

So, we saw DFS and BFS, now we will see how to use these algorithms to be able to define algorithms for these coverage criteria. When I talk about coverage criteria that I discussed in this slide, for each of those coverage criteria, we basically saw two entities. So, one entity was what is called a test requirement. So, when I say you achieve edge-pair coverage the test requirement is achieving edge-pair coverage. Corresponding to this test requirement I look at the graph and generate a set of test paths in the graph that will achieve this test requirement for edge-pair coverage.

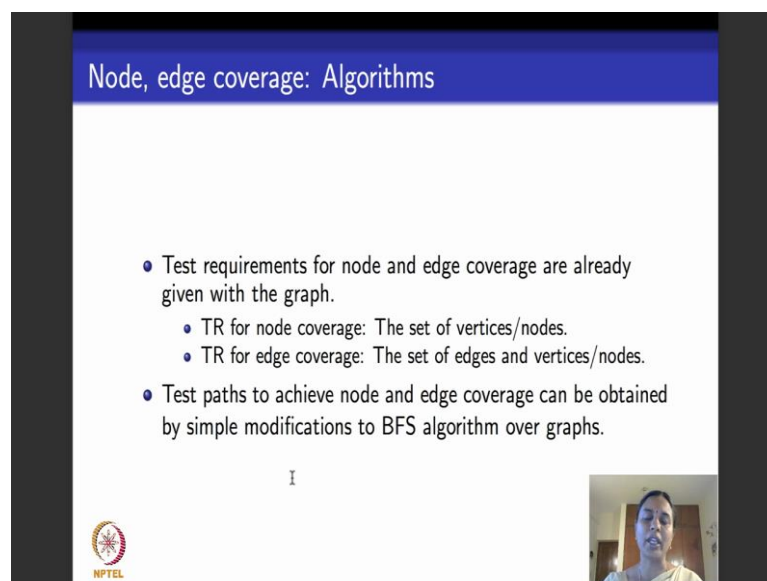
So, when I talk about coverage criteria, I am talking about two entities that are related to the coverage criteria: one is the criteria specified as a test requirement, abbreviated as TR, and the other is the set of test paths which occur a test case that are designed to satisfy the criteria. Now it could very well be the case the criteria that I define is infeasible. Like I told you right, if there is a graph that has a loop and my criteria say do complete path coverage, it is directly infeasible, because there are an infinite number of paths. So, only for feasible coverage criteria as test requirements do we talk about

designing test paths as test cases for them. For infeasible coverage criteria we do not even consider test paths.

So, another interesting problem if you step back and ask, you could ask yourself- am I considering the problem of deciding whether a given coverage criteria is feasible or not? Yes, this is a very important problem, but that will not be the focus of this course, because as I told you it is an undecidable problem to check whether several different coverage criteria are feasible or not. And because this course is intended to be an application oriented course I really do not want to introduce how you get those undecidability results and what are the reduction proofs and all that.

So, what we will focus is; we will assume that the test criteria that we have working with are feasible and then we will see how to write test requirements for them, and how to design test cases as test paths for them.

(Refer Slide Time: 05:39)



The slide is titled "Node, edge coverage: Algorithms" in a blue header. It contains the following bullet points:

- Test requirements for node and edge coverage are already given with the graph.
 - TR for node coverage: The set of vertices/nodes.
 - TR for edge coverage: The set of edges and vertices/nodes.
- Test paths to achieve node and edge coverage can be obtained by simple modifications to BFS algorithm over graphs.

At the bottom left is the NPTEL logo, and at the bottom right is a small video inset showing the speaker.

We will begin with this listing, one at a time we look at. So, we begin with node and vertex coverage and then we will move on around this list. So, we begin with node and edge coverage. As I told you for each of this coverage criteria two things to look at: what is the test requirement or TR and what is the test case that will satisfy those Trs? What are test cases? Test cases are test paths in the graph that will satisfy the test requirement.

Just to recap, a test path always has to begin at a designated initial node and end in one of the designated final nodes. That is one of the requirements that we impose is the part of the definition of test paths in this course. So, for node and edge coverage the test requirements are already given along with the graph. What is the test requirement for node coverage? It is basically the set of all nodes of the graphs, the set of all vertices of the graph? What is the test requirement for edge coverage which is basically the set of edges of the graph? There is nothing much to do.

Now, these are the TRs, they are already given to you, we do not need any algorithms to list them or define them specifically. Now what about test cases or test paths that will satisfy node coverage or edge coverage. If you see for node coverage very simple application of breadth first search that we saw about a lecture, one lecture ago will directly give us node coverage or even depth first search. You take a graph fix a source node and do BFS or DFS on that node, it will result in a breadth first tree or a depth first tree of the graph.

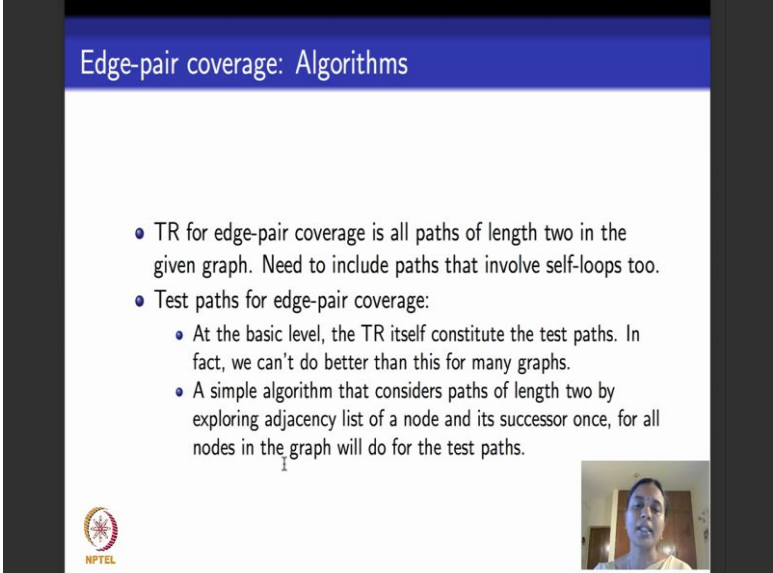
Suppose this tree spans out all the graphs; there it is, the paths of the tree will give you the test path is test cases for node coverage. Suppose such the first breadth first search of the first DFS does not finish exploring all the vertices of the graph; there are vertices that were not reachable from this designated source. Then you fix a new source and start DFS or BFS again from that source and you get another tree. So, this way you run DFS or BFS on the complete graph till you get forest of trees and the resulting forest of trees will give you the set of test paths that will achieve node coverage.

We can do a very similar thing for edge coverage. I can do a simple modification of the BFS algorithm wherein by, after running BFS algorithm I check if I have indeed covered all the edges of the graphs. And if I have not covered all the edges of the graph, I consider the remaining edges which could be cross edges tree edges or back edges and see how I can include them in the test paths to be able to obtain test paths that act as test cases for node and edge coverage.

Just one additional point here. It would be useful to begin your search from the designated initial node in the graph, because that is where ideally a test path will originate. And if you begin your BFS or DFS from the initial node you directly have a test path that will end in one of the final nodes. In case the path does not end in one of

the final nodes we could see how to extend it to the final nodes in case the final node is reachable. So, final node is not reachable from the initial node then you have to begin somewhere in another fresh source for DFS or BFS.

(Refer Slide Time: 08:46)



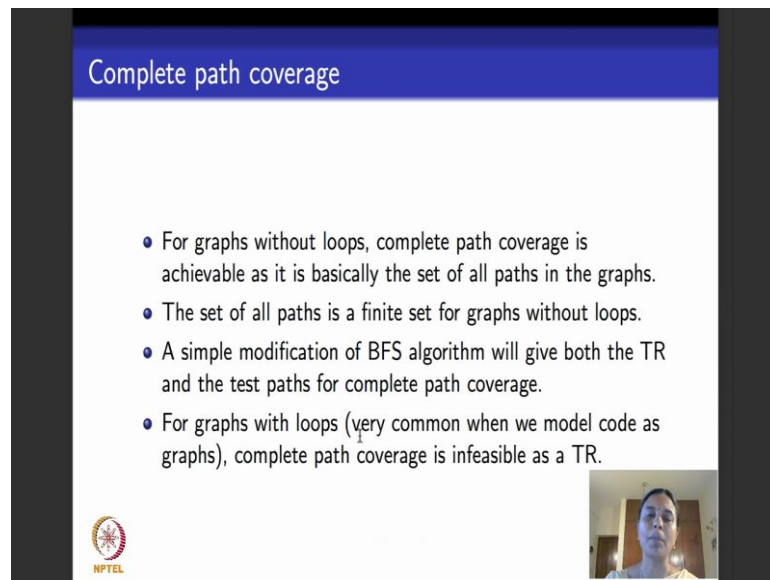
The slide is titled "Edge-pair coverage: Algorithms" in a blue header. It contains a bulleted list of test requirements and paths for edge-pair coverage. In the bottom left corner, there is a small NPTEL logo. In the bottom right corner, there is a small video inset showing a person speaking.

- TR for edge-pair coverage is all paths of length two in the given graph. Need to include paths that involve self-loops too.
- Test paths for edge-pair coverage:
 - At the basic level, the TR itself constitute the test paths. In fact, we can't do better than this for many graphs.
 - A simple algorithm that considers paths of length two by exploring adjacency list of a node and its successor once, for all nodes in the graph will do for the test paths.

We move on: now the next coverage criteria in our list is what is called edge-pair coverage. The test requirement for edge-pair coverage is all paths of length 2. So, here when I say all path of length 2 I need to include paths that involve self loops also. How do I compute all paths of length 2? I can compute all paths of lengths 2 by exploring the adjacent list of a node and its successor and stopping at that. Exploring the adjacency list of one node will give me the edges that are adjacent to that node, that are incident on that node. And exploring the adjacency list of each of these success would get me the edges that are incident each of these successors. This should give me a path of length 2 and that is what is needed for a edge-pair coverage.

Once I have the TR for edge-pair coverage, the test requirement itself could act as a test path. We saw an example of a graph there in fact we cannot do better than that, the TR itself becomes a test path. Otherwise what I do is, I can again run BFS or DFS and get as set of test path that will satisfy edge-pair coverage as a test requirement.

(Refer Slide Time: 09:58)



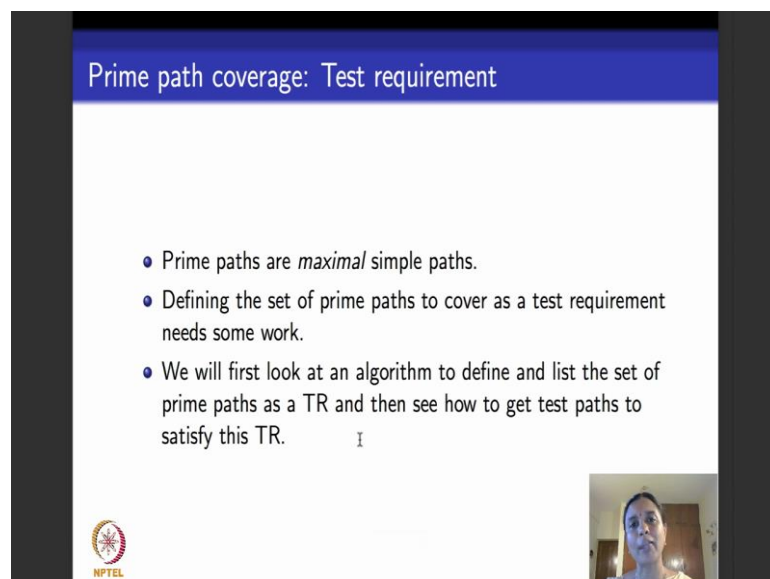
Complete path coverage

- For graphs without loops, complete path coverage is achievable as it is basically the set of all paths in the graphs.
- The set of all paths is a finite set for graphs without loops.
- A simple modification of BFS algorithm will give both the TR and the test paths for complete path coverage.
- For graphs with loops (very common when we model code as graphs), complete path coverage is infeasible as a TR.

NPTEL

Now, the next coverage criterion in our list is what is called complete path coverage. As I told you complete path coverage is many times not feasible, it is not feasible in graph that have loops. And graphs that model software artifacts do have loops. Typically control flow graphs do have loops. So, there is no point in looking at complete path coverage because it will be infeasible, so we directly move on and look at prime path coverage.

(Refer Slide Time: 10:22)



Prime path coverage: Test requirement

- Prime paths are *maximal* simple paths.
- Defining the set of prime paths to cover as a test requirement needs some work.
- We will first look at an algorithm to define and list the set of prime paths as a TR and then see how to get test paths to satisfy this TR.

NPTEL

So, just to recap what is prime path? Prime paths are maximal simple path that is there a simple they do not come as sub path of any other simple path. Now, I want to be able to do the same old two things for prime path, I want to be able to define TR test requirement for prime path, after I finish doing that I want to be able to define test cases that will need this test requirement for this prime path.

To begin with we look at test requirements for prime path and what are the algorithms to see that. So, instead of giving you the pseudo code for the algorithm like we did for BFS and DFS, what I thought we could do is we could take an example graph and I will walk you through how the algorithm will run to compute the list of prime paths. Once the algorithm computes the list of prime paths what I have with me is basically still my test requirements. With this list of prime paths as my test requirement, I now have to go ahead and generate test paths that will cover or satisfy all these prime paths. So, I will work you through an example and explain the algorithm by using that example.

(Refer Slide Time: 11:41)

Computing prime paths: An example

Consider the graph below:

```

graph TD
    Start(( )) --> 0((0))
    0 --> 1((1))
    1 --> 2((2))
    1 --> 5((5))
    2 --> 3((3))
    3 --> 1
    4((4)) --> 4
    4 --> 6(((6)))
    5 --> 6
    style Start fill:none,stroke:none
    style 6 stroke-width:4px
    
```

- Our algorithm will enumerate all simple paths, in order of increasing length.
- We will then choose the prime paths from this list, as and when we enumerate the simple paths.

◀ ▶ ⏪ ⏩ 🔍 🔄

So, here is a simple graph. Here it has seven nodes beginning from 0 and ending at 6; 0 is an initial node, 6 is a final node as it marks with a double circle. This graph has branching, it has branching at node 1, it branches into 5 or 2. This graph also has self loop here at node 4 and this graph has a cycle here 1, 2, 3, 1; this is a cycle. So it will be interesting to look at prime paths. As I told you what is a single big use of prime paths? Assuming that this cycle represents a loop prime paths give you a need coverage criteria

that will help you to cover this loop. It will help you to skip the loop, it will help you to execute the loop and its normal operations.

So, you can here you can assume there are two loops: one self loop here and one cycle here. So, we will see how to compute prime paths for this graph by, as a test requirement first and then we will see test cases that will help us to achieve this test requirement. So, what are prime paths? Prime paths are maximal simple paths. So, what is this strategy that my algorithm will follow is the following.

So, what it will do is that it will take this graph and it will enumerate all simple paths one after the other. So, is there a systematic way of enumerating simple paths? Yes, the following is the systematic way of enumerating simple paths that we will follow. We will enumerate simple paths in order of increasing length; that is we will enumerate simple paths of lengths of 0, then will enumerate simple path of length 1, we will enumerate simple path of length 2 and so on. And then when do we end? What is the criteria to end? Please note that we are enumerating simple paths. So, what is the maximum length that a simple path in a given graph can have? Because simple path cannot contain cycle if the maximum length of path in the given graph can have is the number of vertices.

So, our algorithm is guaranteed to stop. So, what we do to begin with, we enumerate simple paths of increasing length, and as and when we enumerate simple paths of increasing length we will mark out some of those paths as being prime paths and pull them out. As and when we mark and pull them out and finish enumerating all the simple paths the final list of marked and pulled out simple paths will be the prime paths that we will obtain as a test requirements.

So, I will show you how that works for this example graph. So, I begin with enumerating simple paths of length 0.

(Refer Slide Time: 14:21)

Computing prime paths: Example

Simple paths of length 0 (7 paths).

- Path [0]
- Path [1]
- Path [2]
- Path [3]
- Path [4]
- Path [5]
- Path [6]!

Exclamation mark (!) after path [6] implies [6] cannot be extended further. Note that 6 is a final node and has no out-going edges.

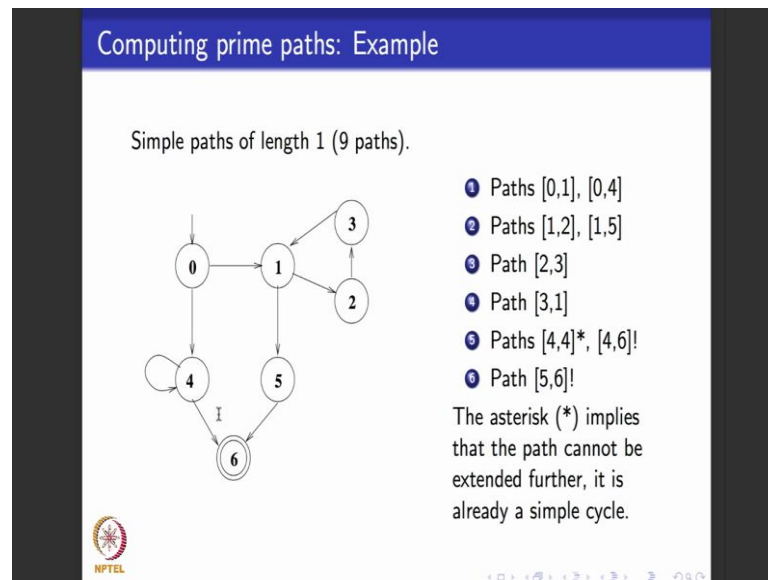
NPTEL

So, what is a simple path of length 0? Simple path of length 0 is just the vertex, because it has one single vertex. How many vertices are there in this graph? There are seven vertices starting from 0, 1, 2, 3 and so on up to 6.

So, I enumerate each of these vertices as a path of length 0. There are seven such paths of length 0. I have just enumerated them. Another thing to be noted is at the end of this 0 length simple path which contains the single vertex 6, I have put an exclamation mark. What is the role of the exclamation mark? The exclamation mark tells us that this path, this simple path which contains the single vertex 6 cannot be extended anymore as far as this graph is concerned. Why is that so? Because 6 is a final vertex and there are no edges that are going out of 6.

So, as far as this graph is concerned, simple path 6 cannot be extended anymore. So, I remember that by putting an exclamation mark. Now, after enumerating all paths of length 0, I go ahead and enumerate paths of length 1.

(Refer Slide Time: 15:34)



So, here is the same graph and here are the paths of length 1. How have I enumerated paths of length 1? I go back to the previous slide. So I say here is a path of length 0 which begins at vertex 0. So, you go back and look at the graph vertex 0; vertex 0, what are the two paths emerging from vertex 0? A path of length 1 from 0 to 4, a path of length 1 from 0 to 1; that is what I have written here vertex 0 this path of length 0 can be extended to two paths of length 1, namely the edge (0, 1) and the edge (0, 4).

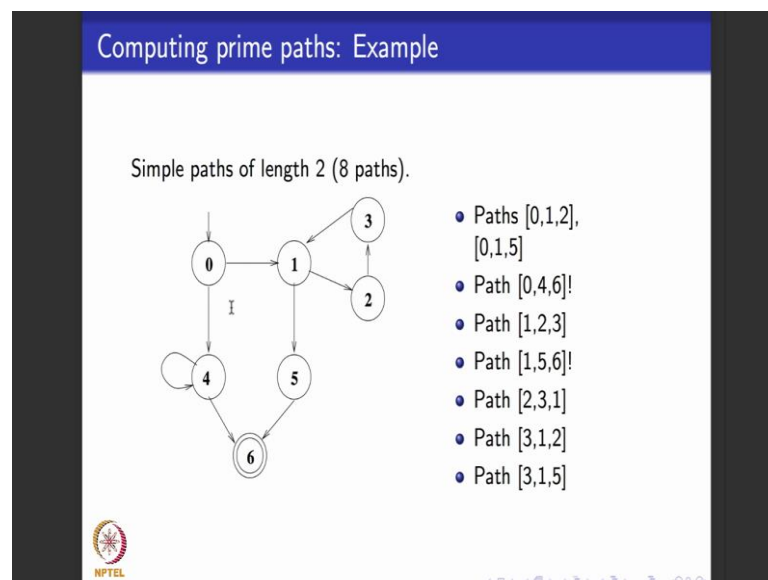
Similarly, vertex one which was a simple path of length 0 can be extended to two paths of length 1 paths 1, 2 and path 1, 5. If we move on vertex 2 can be extended to path 2, 3; vertex 3 can be extended to path 3, 1; vertex 4 can be extended to two simple paths of length 1, one leading to 6 and 1 using this self loop from 4 to itself and vertex 5 can be extended to one simple path of length 1 which is the edge (5, 6).

Now let us look at what are these exclamation marks that we have put like last slide. So, again in this listing of paths of length 1; nine simple paths I have put two paths with exclamation mark, which means both the paths end in the final nodes 6 and they cannot be extended further. So, exclamation mark for us is to mean that this path cannot be extended further. Why am I interested in paths that cannot be extended further? Paths that cannot be extended further could mean that they are heading towards being a maximal simple path. And that is my interest right, maximal simple paths or prime paths is what I want to look at.

So, in addition to exclamation mark I have also gone ahead and put an asterisk here. What is that mean? That means the following; this means that this path which is this edge 4 to 4, it cannot be extended further not because there are no outgoing edges, but because it already forms a simple cycle. Remember if the only way to extended further is to use this edge once again (4, 4) and 4; in which case it will not turn out to be a simple path which is not interest to me. And the other way to do it is to do (4, 4) or 6 in which case also it will not be a simple path so it is not of interest to me. So, I mark it with a asterisk which says that this path also cannot be extended further.

Now I look at all the other remaining paths which is this list on top, and see how to extend them to get simple path of length 2.

(Refer Slide Time: 18:27)

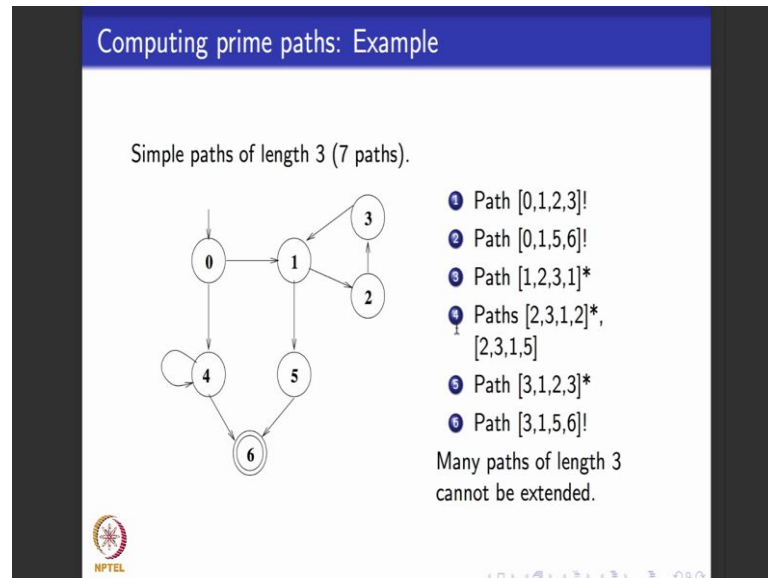


So, you look at the paths 0, 1; I extend the path 0, 1 into two possible paths of length 2 path going from 0 to 1 and then from 1 to 2 and path going from 0 to 1 and then from 1 to 5; that is what is listed here.

Similarly, the path 0, 4 can be extended to 0, 4, 6. Please note that I can also extend 0, 4 to 0, 4 and 4, but that is not as simple path so I do not list it here. And I go on doing this, path 1, 2 can be extended to 1, 2, 3, path 1, 5 can be extended to 1, 5, 6 which comes with an exclamation mark because that is where it stops, cannot be extended further. Similarly path 2, 3 can be extended to 2, 3, 1. 3, 1 can be extended to 3, 1, 2 and 3, 1, 5.

So, paths of length 2, how many paths are there? Eight paths are there and two of them cannot be extended further.

(Refer Slide Time: 19:24)

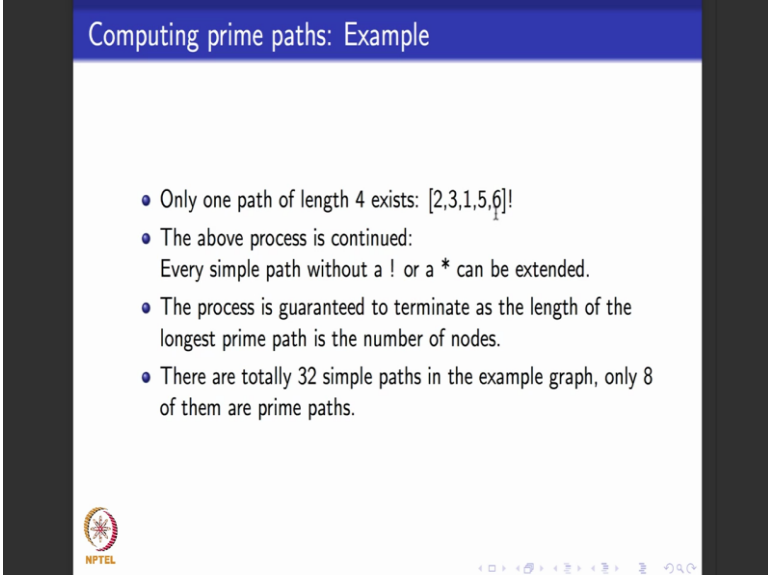


Now, I start with paths of length 2 and consider path of length 3. It so happens that I get seven different paths. So, path 0, 1, 2 can be extended to 0, 1, 2, 3. Path 0, 1, 5 can be extended to 0, 1, 5, 6 right, path 1, 2, 3 can be extended to 1, 2, 3, 1. I can go back and so on.

If you see in this listing almost every paths of length 3 is marked with an exclamation mark or an asterisk. In fact, there is exactly one path that is not marked with either of this. What do all these marked paths mean? They mean that that is it, we cannot extend them anymore. We cannot extend them anymore for two reasons: one is they end in the vertex from which there are no outgoing edges or if we extend them further, I violate the criteria of they being a simple path.

So, I am pretty much closed here as far as paths of length 3 are concerned. There is only one path here 2, 3, 1, 5 that is not marked, so that is the only path that has go for extension. So, I go ahead take that path and extend it by adding the edge (5, 6). You see 2, 3, 1, 5 that is a path of length 3 I can extend it to length 4 by adding this edge (5, 6).

(Refer Slide Time: 20:36)



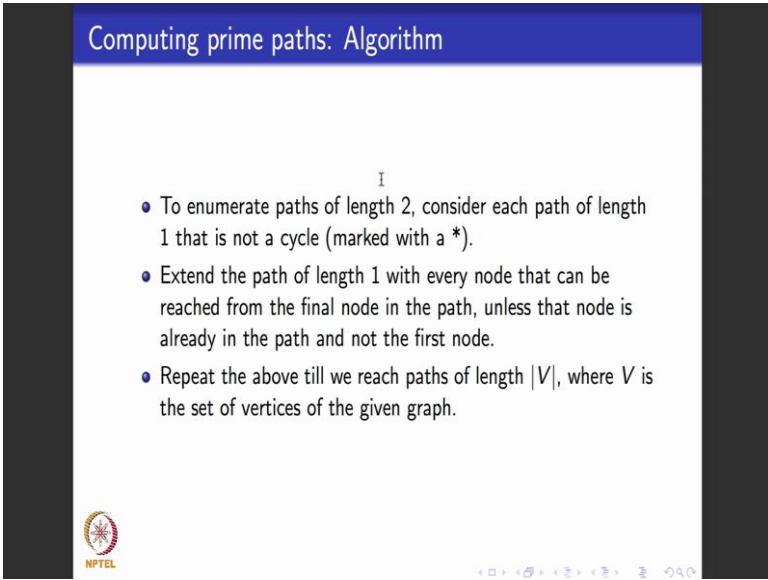
Computing prime paths: Example

- Only one path of length 4 exists: $[2,3,1,5,6]!$
- The above process is continued:
Every simple path without a ! or a * can be extended.
- The process is guaranteed to terminate as the length of the longest prime path is the number of nodes.
- There are totally 32 simple paths in the example graph, only 8 of them are prime paths.

NPTEL

That is what I have done here. And the moment I did that I am forced to put an exclamation mark here for the same reason, because I have ended in the node 6 which is a final node and it does not have any outgoing edges.

(Refer Slide Time: 21:05)



Computing prime paths: Algorithm

I

- To enumerate paths of length 2, consider each path of length 1 that is not a cycle (marked with a *).
- Extend the path of length 1 with every node that can be reached from the final node in the path, unless that node is already in the path and not the first node.
- Repeat the above till we reach paths of length $|V|$, where V is the set of vertices of the given graph.

NPTEL

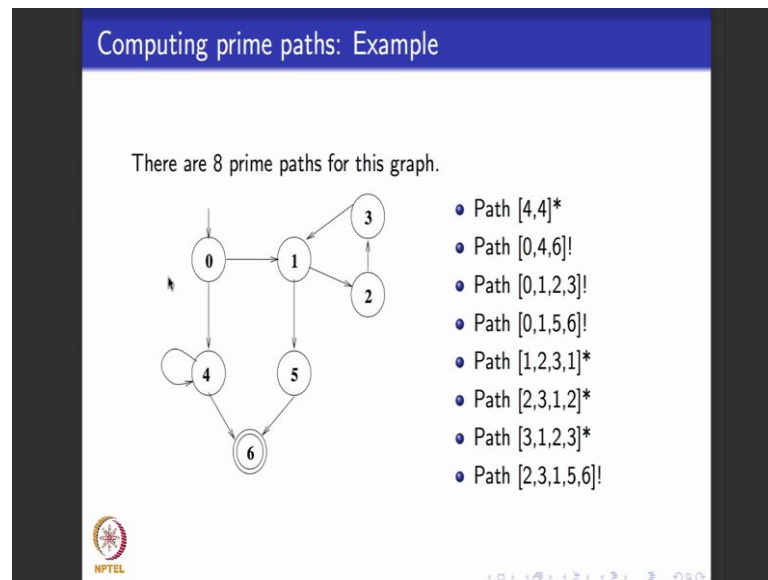
So, what do I do? Here is the algorithm in English. I consider paths of length 2 by extending paths of length 1 with every node that can be reach from the final node in the path unless that node is already in the path and is not the first node. And I keep doing this for paths of length n , increase to path of length n plus 1 till I reach path of length

mod V . Why do I stop at mod V ; because that gives me the maximum length simple paths. Any other path of length greater than mod V , by pigeon hole principle, one vertex as to repeat and the path will not be simple any more.

So, I hope the algorithm is clear. What is the algorithm do? Starts with paths of length 0 extends them to obtain paths of length 1, extends them to obtain paths of length 2, extends them to obtain paths of length 3, keeps repeating this still it can get paths of maximum length which is the length as the number of vertices. While doing this extension it marks out certain paths. There are two kinds of markings that we do; we mark at with an exclamation mark or we mark with an asterisk. When do we mark with an exclamation mark? When I know that that particular path cannot be extended because there may not be any outgoing edges. I mark it with an asterisk when I know that if I extended I will violate the criteria that this path needs to be a simple path. All the other paths at every step that are not marked with one of the special markers can be extended and I keep repeating this process.

So, for this particular graph it so happens that there are 32 simple path for this particular graph, but only eight of them are prime paths. And those are the ones that have been obtained by these markings. In general it is not an easy problem to count the number of simple path, I have just given this for the specific graph. And why is this algorithm guaranteed to terminate, because it is to stop when I get a path of length mod V as we discussed.

(Refer Slide Time: 23:07)



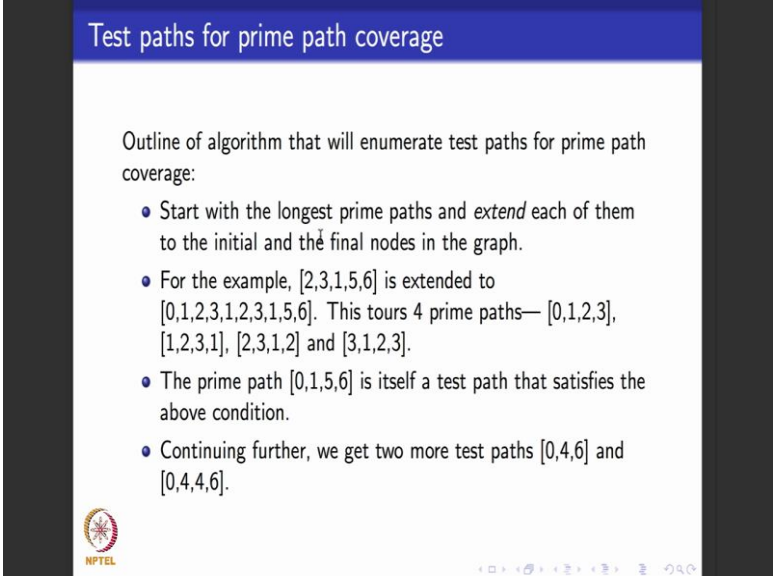
So, now I will just present this graph and all the prime paths for the graph. So, this is the graph that we have been using to understand an algorithm, and here are all the prime paths that I worked my algorithm out and I will listed through that. So, let us look at them :the path 4, 4 is this which was listed as a path of length 1 if you remember and right then marked with the star because it could not be extended it was already a simple cycle. Then there was a path 0, 4, 6. Then there is a path 0, 1, 2, 3; 0, 1, 5, 6; then 1, 2, 3, 1; and then 2, 3, 1, 2; 3 1, 2, 3.

If you see these three paths no 1, 2, 3, 1 they basically corresponded this cycle in the graph and between the three paths they tell you various ways in which you could traverse this loop. And this path 4, 4 is a prime path that covers this loop. The rest of the paths traverse the rest of the graph. Like for the example the path 0, 1, 5, 6 skip this loop, the path 0, 4, 6 skips this loop. And then the other path that is left is 2, 3, 1, 5, 6 that is like exiting this loop 1, 2, 3 and coming out to a final node.

So, for this graph, our algorithm that begins by enumerating simple path of increasing length and identifying prime path as and when we enumerate them, ends by listing all these eight paths as prime path in the graph. And if you sort of play them back in the graph like we did just now you can see how they neatly cover the two loops and how they skip the loop in the graph.

Now, what have we done? What we have done so far just gives us the test requirement or TR for prime path coverage. We say now you write a set of test cases for covering these paths. This elaborate exercise that we did was just to obtain the test requirement for prime path coverage. We still have to go ahead and design test cases to be able to get test paths to meet the test requirement for prime path coverage.

(Refer Slide Time: 25:28)



Test paths for prime path coverage

Outline of algorithm that will enumerate test paths for prime path coverage:

- Start with the longest prime paths and *extend* each of them to the initial and the final nodes in the graph.
- For the example, [2,3,1,5,6] is extended to [0,1,2,3,1,2,3,1,5,6]. This tours 4 prime paths— [0,1,2,3], [1,2,3,1], [2,3,1,2] and [3,1,2,3].
- The prime path [0,1,5,6] is itself a test path that satisfies the above condition.
- Continuing further, we get two more test paths [0,4,6] and [0,4,4,6].

NPTEL

How do we go ahead and get the test paths for prime path coverage? Again there are several algorithms that have been used in literature. I will give you one algorithm that I will illustrate on the same example because that turns out to list the prime paths. It is a heuristic that lists is reasonably faster than other known algorithms. So, what we do is in this list of prime path which are given as my test requirement you start with the longest path. Which is the longest path in this list? That is the last path: 2, 3, 1, 5, 6. What you do is that this you extend this path to the left to see if you can find a path from the initial node to the beginning node of this path. And then from the right assuming that this is not a final node you see if you can extend it to the right to make it end in a final node.

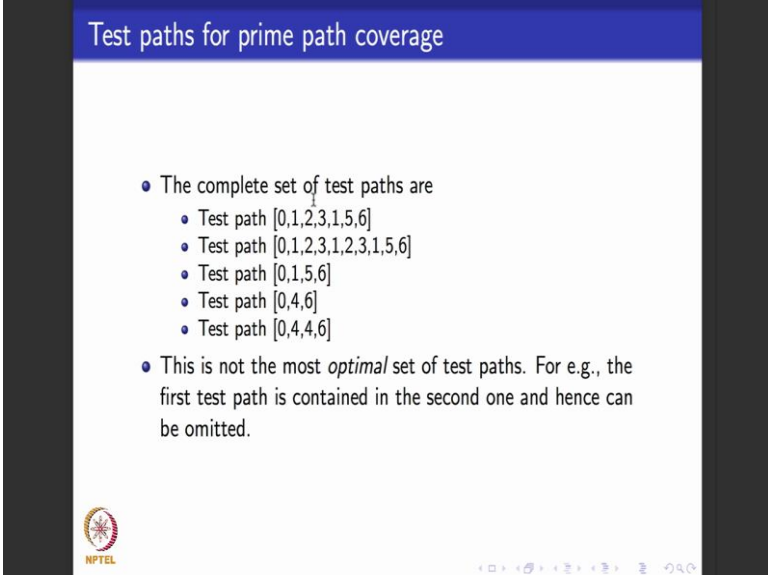
In this case 6 is already a final node, so I have already ended in the final node, but 2 is not an initial node. So, I take this longest prime path and see if I can go to the left and extend this path to make it as if it was beginning from an initial node. So, if I do that for 2, 3, 1, 5, 6, I get such a path; I get 0, 1, 2, 3, 1, 2, 3, 1, 5, 6. So, I will trace it out here 0, 1, 2, 3 right; 1, 2, 3, 1, 5, 6. Now you might ask why did I repeat this 1, 2, 3 twice? I

repeated that with the purpose of I could have not done that, I could have done 0, 1, 2, 3, 1, 5, 6 in which case I would not have covered these paths here. If you remember I had three prime paths which tell you how to traverse the loop beginning from three different vertices of the loop. I could begin the loop at 1, I could begin the loop at 2, I could begin the loop at 3; then you basically going round the loop once, but the vertices in which they were beginning were different. So, that I want to be able to cover all of them here so that is the reason why I repeat this loop here.

Please note that this is a test path, so that need not satisfy the notion of a simple path or a prime path, because it is just an ordinary test path. The only condition that it has to satisfy is that it has to begin at an initial node which is 0 for our example and end at a final node which is 6 for an example. In between nodes can be repeated, because it is not a test requirement for prime path coverage it is only a test path for prime path coverage. So, test paths need not be simple cycles, they need not be maximal simple paths. So, if I do that then right there I have toured four prime paths in this list: I have done this, I have done this loop, so I have done I entered the loop from the initial state and I have gone through the loop.

So, what I do is the remaining? What is the remaining? Amongst the remaining prime path which is the longest prime path that is left out that is this 0, 1, 5, 6. It so happens that that already begins in an initial node and ends in final node. So, does have to be extended anymore, so I leave it. What is the other path that is left out? Those are these two. In this path 0, 4, 6 again begins in the initial node and end in the final node, but I need to be able to give scope to cover this loop so I extended it to 0, 4, 4, 6.

(Refer Slide Time: 29:13)



The slide is titled "Test paths for prime path coverage" in a blue header. It contains a bulleted list of test paths and a note about their optimality. The list includes:

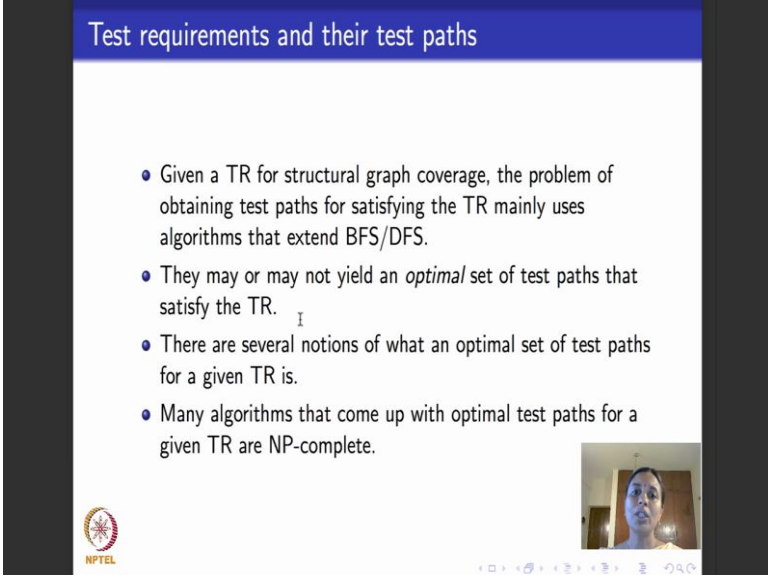
- The complete set of test paths are
 - Test path [0,1,2,3,1,5,6]
 - Test path [0,1,2,3,1,2,3,1,5,6]
 - Test path [0,1,5,6]
 - Test path [0,4,6]
 - Test path [0,4,4,6]
- This is not the most *optimal* set of test paths. For e.g., the first test path is contained in the second one and hence can be omitted.

The NPTEL logo is visible in the bottom left corner of the slide.

So, putting it all together the complete set of test paths for prime path coverage as my test requirement are this list that I obtain. If you remember I took this which is the longest prime path I extended it to cover the loop over the vertices 1, 2 and 3, then I took the path that skipped the loop, then there was one more loop which these two test paths cover. If you notice and if you worry about is this the best set of test paths that I can get for prime path coverage for this graph, I would say no. Why because, if you see the simple reason the 0, 1, 2, 3, 1, 5, 6 is completely embedded within this second path, so I could remove one of them. So, I have not got the least number of test path that I could cover.

Similarly 0, 4, 6 is completely embedded in 0, 4, 4, 6, so I could remove 0, 4, 6, but that is alright. The focus for us is not to get the most optimal set of test paths, the focus for us is to be able to correctly define the test requirement and get a set of test paths that satisfy the test requirements; that is what we have done here.

(Refer Slide Time: 30:22)



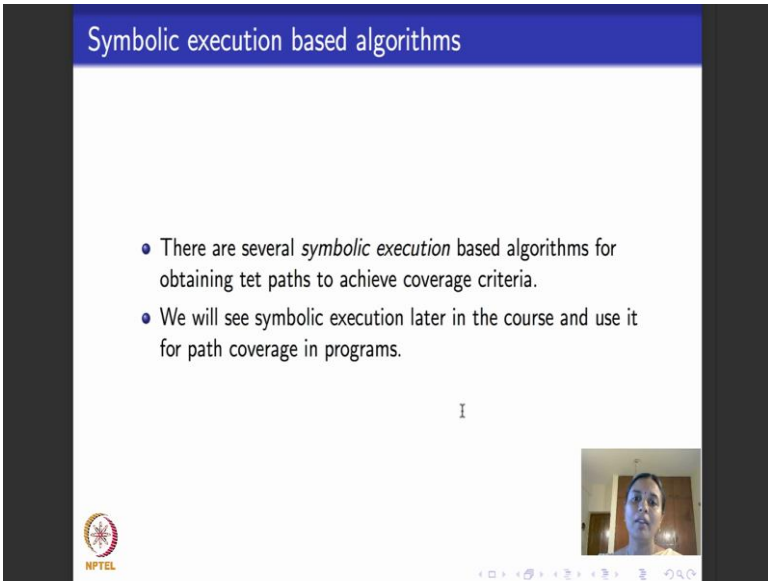
Test requirements and their test paths

- Given a TR for structural graph coverage, the problem of obtaining test paths for satisfying the TR mainly uses algorithms that extend BFS/DFS.
- They may or may not yield an *optimal* set of test paths that satisfy the TR.
- There are several notions of what an optimal set of test paths for a given TR is.
- Many algorithms that come up with optimal test paths for a given TR are NP-complete.

NPTEL

In general the problem of what is optimality, optimal test path corresponding to a test requirement. There are several notions of optimality available in the literature. We will not really focus on that because I want to be able to move on, but I point you to good references where you could look up for this kind of information. In general many algorithms that come up with optimal test paths, or decide what is an optimal notion are typically intractable problems and lot of them are NP-complete problems.

(Refer Slide Time: 30:53)



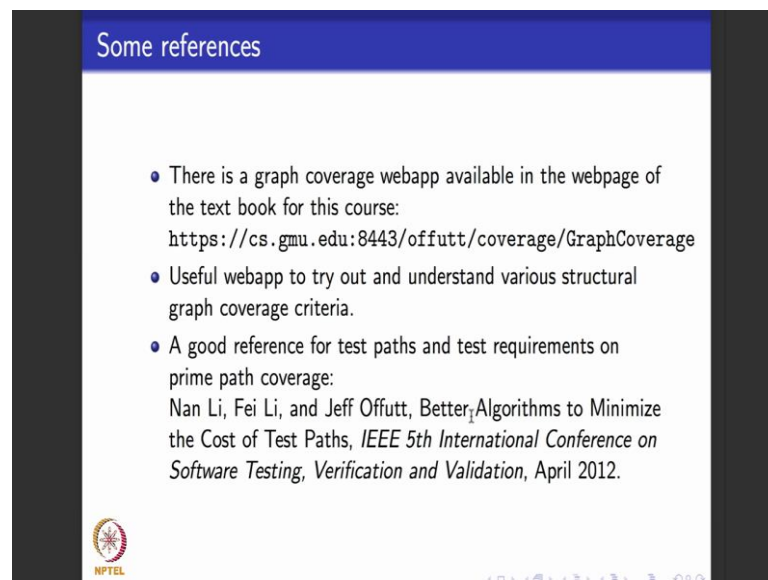
Symbolic execution based algorithms

- There are several *symbolic execution* based algorithms for obtaining test paths to achieve coverage criteria.
- We will see symbolic execution later in the course and use it for path coverage in programs.

NPTEL

In fact, what we saw was explicit state algorithms- algorithms that use breath first search or depth first search. You could leave them aside and do what is called symbolic execution based algorithms. I will do symbolic execution a little later in the course after a few weeks; they also can be used to obtain specified path coverage criteria.

(Refer Slide Time: 31:15)



I would like to end this module by encouraging you to use these web apps. So, the text book that have been using for this course is the book by Ammann and Offutt called software testing in the web page of the text book they have a very nice web app for several different coverage criteria that they introduce in the textbook. For now you begin by using this particular web app, the app that is available in this URL. You can try out how the various structural coverage criteria work by input in your own graph. And then they have algorithms which are basically Java programs they have written which will output the test requirement and the test paths that satisfy the test requirement.

If you want go ahead and read further on, this paper is a very good reference to talk about prime path coverage, which is the most difficult structural coverage criteria that we saw till now.

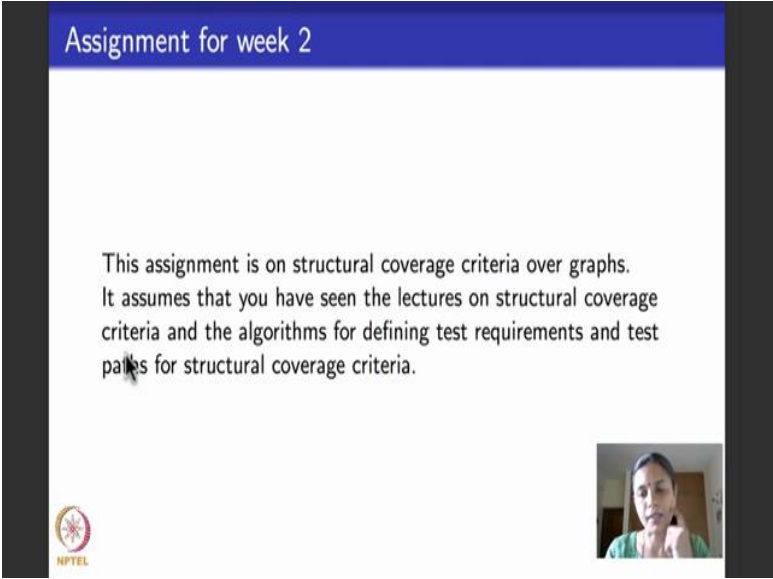
Thank you.

Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 10
Assignment 2: Structural Coverage Criteria

Hello everyone. So, this is the first lecture of third week. What I wanted to do today that is not really a lecture, but more to help you walk through the second week's assignment, assuming that you have looked at it and you made an attempt to solve it and you have also uploaded solutions. We will see how we will go about solving it, what are the questions, let us discuss our understanding of structure and coverage criteria over graphs to see if we could solve this assignment fully, correctly.

(Refer Slide Time: 00:43)



Assignment for week 2

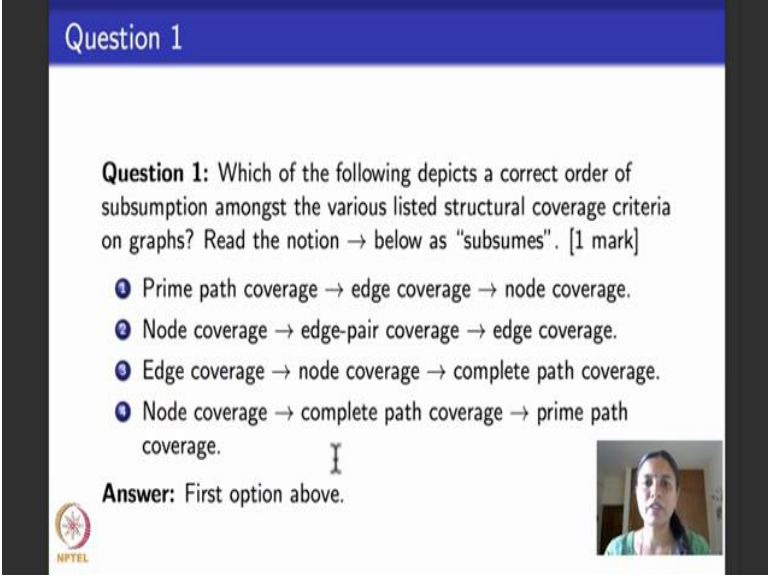
This assignment is on structural coverage criteria over graphs. It assumes that you have seen the lectures on structural coverage criteria and the algorithms for defining test requirements and test paths for structural coverage criteria.

NPTEL

The slide features a blue header with the title 'Assignment for week 2'. The main content area is white with black text. In the bottom right corner, there is a small video inset showing a woman (Prof. Meenakshi D'Souza) speaking. The NPTEL logo is visible in the bottom left corner of the slide.

This lecture assumes that you have gone through all the videos of the graph structural coverage criteria, the algorithms for graph structural coverage criteria, and you have also looked at the assignment for the second week and made an attempt to solve it. So, what I will do now is I will walk you through one question after the other, tell you what the correct answer is and tell you how to get the correct answer.

(Refer Slide Time: 01:06)






Question 1

Question 1: Which of the following depicts a correct order of subsumption amongst the various listed structural coverage criteria on graphs? Read the notion \rightarrow below as "subsumes". [1 mark]

- ❶ Prime path coverage \rightarrow edge coverage \rightarrow node coverage.
- ❷ Node coverage \rightarrow edge-pair coverage \rightarrow edge coverage.
- ❸ Edge coverage \rightarrow node coverage \rightarrow complete path coverage.
- ❹ Node coverage \rightarrow complete path coverage \rightarrow prime path coverage.

Answer: First option above.

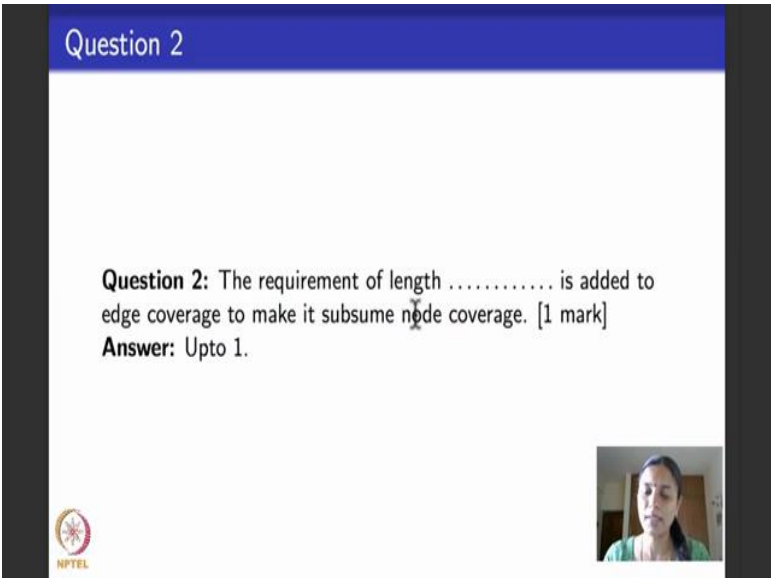


So, which was the first question? The assignment had 8 questions, the first question talked about coverage criterion subsumption; to read out the question it asks which of the following depicts a correct order of subsumption amongst the various listed coverage criteria below. So, this notion, this symbol right arrow you should read it as subsumes. So, how do I read the first option? Your answer is to choose one of these options, the correct option. How do I read the first option? You read it as prime path coverage subsumes edge coverage which in turn subsumes node coverage. Similarly second one would be node coverage subsumes edge pair coverage which in turn subsumes edge coverage and so on? So, the correct choice for this answer for this question happens to be the first answer, that is this one. Prime path coverage that subsume edge coverage which in turn subsumes node coverage.

Now, why are the rest of them not the correct option? If you see the second option it says node coverage subsumes edge pair coverage which in terms subsumes edge coverage. So, it as to subsumption relation like all other options this the second one edge pairs coverage subsuming edge coverage is correct, but node coverage does not subsume edge pair coverage. So, this is not the correct option similarly for the third one, edge coverage does subsume node coverage, but node coverage clearly does not subsume complete path coverage. In fact, complete path coverage could even be infeasible as we discussed several times.

So, third one also cannot be the right option. What is the fourth one? It says node coverage subsumes complete path coverage which in terms of subsumes prime path coverage. So, this part is correct, complete path coverage that subsumes guide path coverage because if I do complete paths I always include prime paths in my TR, but this is not correct as I told you in the option for the previous answer node coverage does not subsume complete path coverage. So, except for 1 the other options 2, 3 and 4, one of the listed subsumption is not correct. So, none of them can be a correct answer. In the first one both the listed subsumption criteria happened to be correct. So, the first one is the correct answer.

(Refer Slide Time: 03:28)



Question 2

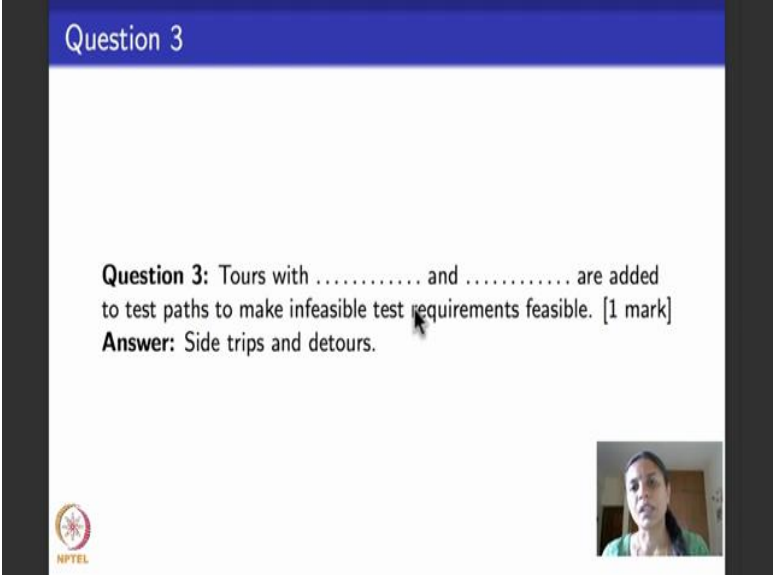
Question 2: The requirement of length is added to edge coverage to make it subsume node coverage. [1 mark]
 Answer: Upto 1.

NPTEL

Now, moving on to the second question. Second question was a simple fill in the blank question, it asks the following it says the requirement of length dash should be added to edge coverage to make it subsume node coverage. The answer is requirement of length up to 1, if you remember what edge coverage is, the test requirement or t r for edge coverage said you cover all the edges of the graph. So, all the edges of the graph means all paths of length exactly 1. So, I do not keep it as exactly one I make it as a length up to 1 because I want edge coverage to subsume node coverage. Node coverage TR says nodes the path of length 0. So, if I include the requirement length up to 1 it includes paths of length 0 and it also includes paths of length what. So, because I want edge coverage to subsume node coverage, I change the requirement for edge coverage to be all

parts of length up to 1 as it is test requirement. So, the answer for this would be all paths of length up to 1.

(Refer Slide Time: 04:31)



Question 3

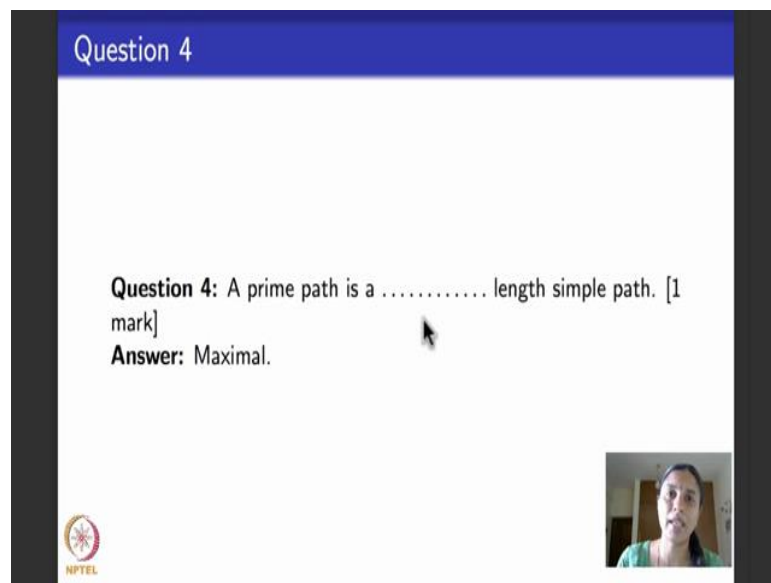
Question 3: Tours with and are added to test paths to make infeasible test requirements feasible. [1 mark]
Answer: Side trips and detours.

NPTEL

The next question asks, again a fill in the blank question, it says tours with the dash and dash are added to test parts to make infeasible requirements feasible. If you remember the lectures we have discussed about side trips and detours. Side trip, basically you take the test paths and then at some point in a vertices test path you decide to move out take another small side trip come back and join the vertex right. What would be a detour? Detour is very similar to a side trip, but it might skip some edges also on the side trip. Why would I need this? If you remember test path especially those dealing with prime path coverage could sometimes become infeasible, the test requirements will become infeasible.

Especially if I have a loop like do while loop and I look for prime path coverage on that do while loop and may not be able to skip the loop at all right because do while says you first do one execution of the loop and then you check for the condition. Whereas prime path coverage will insist on skipping the loop. So, I say I meet that test requirement for prime path coverage by using a test path that will skip the loop, but actually execute the loop as a side trip right. So, they are very useful to make infeasible test paths as feasible.

(Refer Slide Time: 05:56)



Question 4

Question 4: A prime path is a length simple path. [1 mark]

Answer: Maximal.

So, the next question was again fill in the blank question it asks what sort of path a prime path is. The prime path, if you remember the definition is a simple path there does not come as a sub path of any other path. So, prime path are what are called maximal simple path, maximal length simple path. Why do I put maximal and not maximum? I hope you know the difference maximal says that they could be more than one maximum length simple path right. There could be you see the examples if you remember that we looked at while dealing with algorithms for structuring coverage criteria, there is prime path of length 2 there were prime paths of length 3 and there were prime paths of length 4. For each of these lengths that happens to be the maximum length simple path. So, that is why we say maximal. So, the answer to this in a prime path is a maximal length simple path.

(Refer Slide Time: 06:48)

Questions 5, 6 and 7

For the questions 5,6 and 7, we consider the following graph
 $G = (V, E, \{1\}, \{7\})$, with 1 being the initial vertex and 7 being the only final vertex:

- $V = \{1, 2, 3, 4, 5, 6, 7\}$.
- $E = \{(1, 2), (1, 7), (2, 3), (2, 4), (3, 2), (4, 5), (4, 6), (5, 6), (6, 1)\}$.

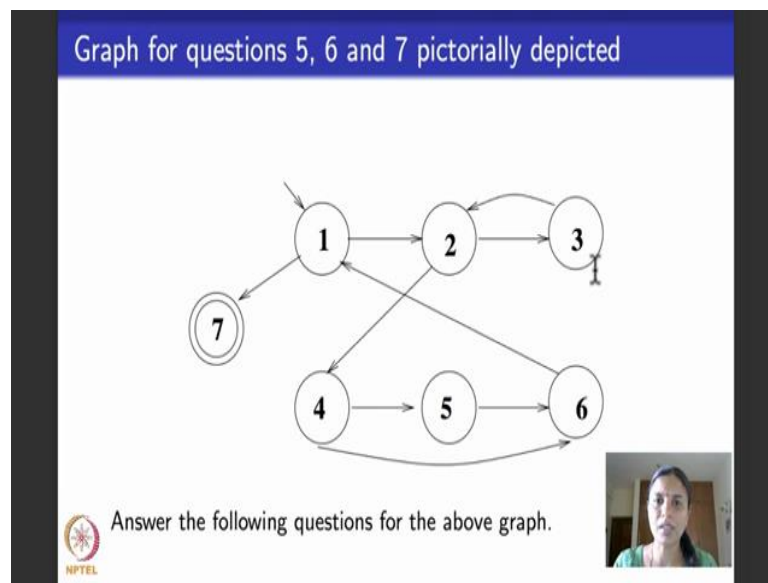
The slide includes an NPTEL logo in the bottom left and a video inset of a woman in the bottom right.

The last 3 questions in the assignment where to deal with coverage criteria that is based on an example graph that was given as a part of the assignment to you. So, here is a graph. So, the graph had 7 vertexes. Numbered 1 to 7, the vertex 1 was the initial vertex or the start vertex the vertex 7 was the only final vertex and these were all the edges in the graph. And then you had 3 questions 5, 6 and 7 which dealt with answering questions about coverage criteria on this graph.

So, before we go ahead and look at how to answer the questions 5, 6 and 7, whenever I am asked a question like this the first wise thing to do is to be able to draw the graphs to be able to visualize the graph for yourself. What are the benefit is of drawing the graph, one is the notion of how the edges are placed what the edges are and how to look for paths in the graph becomes visually and it becomes a little more clear right? Then instead of staring at the set repeatedly to figure out what the notion of path is and things like how to achieve node coverage how to achieve edge coverage even to the extent of how to achieve prime paths coverage can be looked at the visual graph and answered very easily. Intuitively, you do not even have to use the algorithms that we discussed to be able to come up with test requirements and test paths for these criteria.

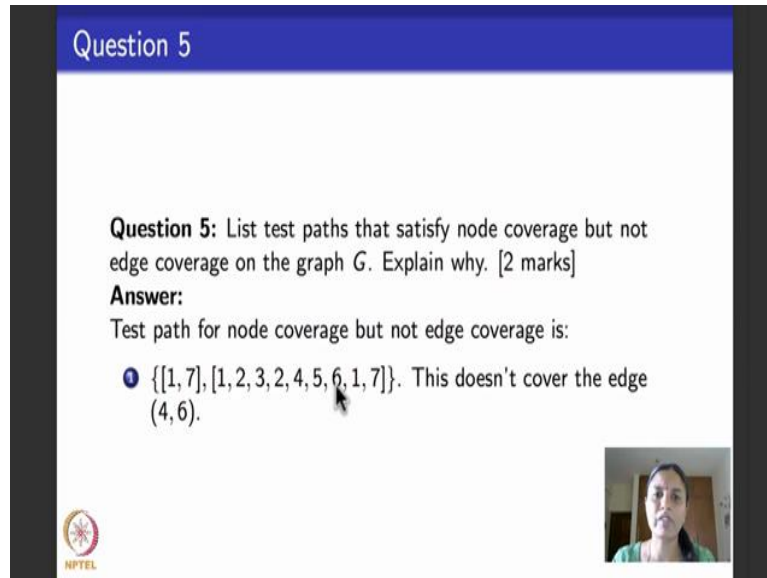
So, that is what I have done for you. Now I have drawn this graph. So, this graph has 7 vertices 1, 2, 3 and so on up to 7.

(Refer Slide Time: 08:17)



So, I have put this 7 vertices. Another thing to note is that while drawing the graph you may not get the layout of the graph. So, perfectly fine do not worry about that if you have to draw long edges, short edges, queued edges you still it is still helps to have a visual graph. And then a mark this initial vertex one mark the final vertex 7 using double circles, and then I have marked the edges if you see there were so many edges about 9 edges. So, there were 2 edges going out of 1, 1 to 2 and 1 to 7, I have marked those edges 1 to 2, 1 to 7, 2 edges is going out of 2, 2 to 3, 2 to 4, marked here 2 to 3, 2 to 4 only 1 edge going out 3 which is from 3 to 2, 3 to 2. Two edges going out of 4, one to 5, one to 6, 4 to 5, 4 to 6; one edge going out of 5 to 6 and 1 edge going from 6 to 1. So, here is one edge going from 5 to 6 one edge going from 6 to 1. So, I think we captured all the edges here. So, this is the graph that is the same as this, but drawn.

(Refer Slide Time: 09:31)



Question 5

Question 5: List test paths that satisfy node coverage but not edge coverage on the graph G. Explain why. [2 marks]

Answer:
Test path for node coverage but not edge coverage is:

• $\{[1, 7], [1, 2, 3, 2, 4, 5, 6, 1, 7]\}$. This doesn't cover the edge (4, 6).

NPTEL

So, now let us go ahead and look at the questions that were asked about this graphs. So, question number 5 says to list all the test paths that satisfy node coverage, but it also says that some of list tests path in such a way that that they satisfy a node coverage, but do not meet edge coverage. If you remember node coverage does not subsume edge coverage in all cases. So, in this particular graph node coverage need not subsume edge coverage. So, it might be possible to come up with a set of test path that satisfy node coverage, but not edge coverage. So, what is node coverage test requirement for this graph? It is basically the set consisting of all the 7 vertices. What are the test paths for node coverage? If you see in this path from 1 to 7, is like a loner path right, it is on it is own. So, you have to be able to take this this is the only way to go to 7. So, to do node coverage for the vertex 7, I need to be able to include this as a standalone test path. It is a test path because it satisfies all the requirements of a test path it begins at an initial vertex which is one it ends in the final vertex which is 7.

So, I have to be able to include this test path to node cover 7. So, the remaining nodes are 2, 3, 4, 5 and 6. So, how do I cover them I can include any test path remember test paths always have to begin in in initial vertex and end in a final vertex. So, I begin at 1, I do not have much of a choice, here I go 2 from 2; let us say I go to 4, 5, 6; from 6 I go to 1. So, which are the nodes I have covered if I had taken such a path ? I have covered 1, 2, 4, 5, 6, earlier I had covered 1 and 7 and left on 3; right, but I have not still not left out. So, I can always go back and then I will continue this path then take it from 1 to 2, I visit 2

once again that is all right there is no harm in this because test paths can have cycles, and then I go to 3. So, I have covered now all the vertices, but I cannot end my test path here because the test path has to end in a final vertex. So, I take this vertex back, and now I still cannot go to 7 the only way to go to 7 is to go to 4 and then may be do this 4 to 6, 6 to 1 and 1 receive.

So, basically what I am trying to say is that if my test requirement is node coverage then there has to be a test path that consists of this segment in the graph which is just this edge 1 to 7. And for all the remaining vertices you need to go through this whole zigzag cycle once and come back to 7. It could include any test path that does that. Here, what I have done is I have included this test path I have this, 1, 2, 3, 2, 4, 5, 6, 1, 7. So, to trace it out 1, 2, 3, 2, 4, 5, 6, 1, 7, it does need node coverage if you see all the nodes that are included in the test path, but I had one more condition in my question that was do not have do not make it subsume edge coverage. So, I am choosing test path in such a way that I have left out one edge which is this edge I have left out the edge 4, 6. So, if you see this test path achieves node coverage, but it almost achieves edge coverage it visits all the edges except for this edge 4, 6. So, this test path is good enough an answer for this question.



(Refer Slide Time: 12:53)

Question 6

Question 6: List test paths that satisfy edge coverage on the graph. [2 marks]

Answer:
Four possible correct answers could be given for this question.
Test paths for edge coverage are

- $\{[1, 7], [1, 2, 3, 2, 4, 5, 6, 1, 7], [1, 2, 4, 6, 1, 7]\}$, or
- $\{[1, 7], [1, 2, 3, 2, 4, 6, 1, 7], [1, 2, 4, 5, 6, 1, 7]\}$, or
- $\{[1, 7], [1, 2, 3, 2, 4, 5, 6, 1, 2, 4, 6, 1, 7]\}$, or
- $\{[1, 7], [1, 2, 3, 2, 4, 6, 1, 2, 4, 5, 6, 1, 7]\}$.

So, the next questions is list all the test paths that can satisfy edge coverage criteria on the graph. So, here they could be several possible correct answers. In fact, 4 different

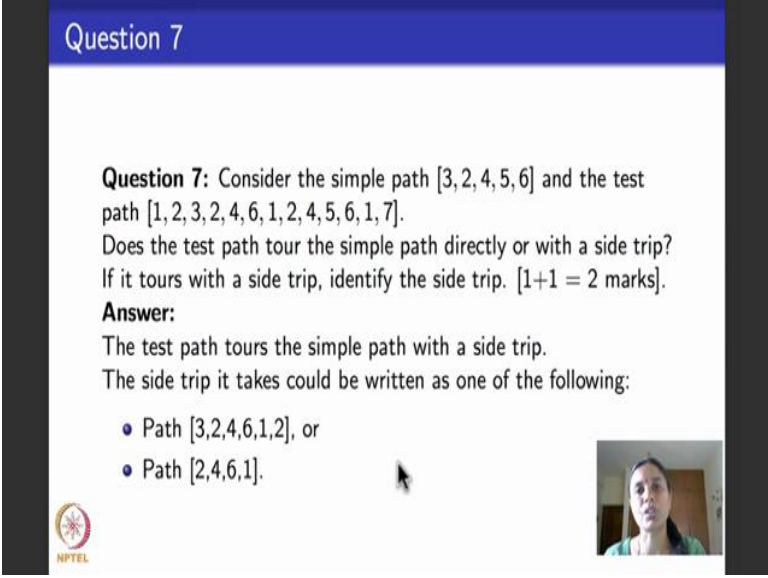
possible correct answers are there. Let us go back and look at the graph. As I told you right, this path is the standalone path it always has to be taken whether you do more coverage edge coverage prime paths coverage. So, 1, 7 will always be there. If you see in all the 4 options that I have listed here 1, 7 is always there and then there is a question of this covering the rest to the edges. So, if you see when I execute the only thing to remember is that all these are pretty much serial, it is only at this place do I have a branching or a choice of course, 2 also I have a branching, but it is this this, but once I do that again I have another choice here at 4.

So, suppose I take this branch 4 to 5 and 5 to 6. Then I have left out this one this edge 4 to 6. So, I need to be able to come back how do I come back only way to come back is to go from 6 to 1, 1 to 2, 2 to 3, 3 back to, 2 to 4 and then I have to take 4 to 6. So, the 4 options basically exploit this. So, you have to do 1, 7 separately, and what are these 4 options? They let you do the let you traverse from 4 to 6 either using the 2 edges 4 to 5, 5 to 6 or using this single edge 4 to 6. So, it is 1, 2, 3, 2; use the edge 4, 5, 6, go back to 1 and then to 7, finish. And then you begin at 1 go to 2 use the edge 4 to 6, the direct edge do 1 and 7 again. This one is the same thing, but it says, so, I will just trace it out here you do 1, 2, 3, 4, 5, 6, 1, 7 for you do 1, 2, 3, 4, 6, 1, 7.

The second one what I have done is, I have done, 1, 2 4, 6, 1, 7 and then I have done, 1, 2, 3, 2, 4, 5, 6, 1, 7. So, those are the 2 options here. And the last 2 options I have basically joined them both. I have avoided the repetition of going back to 7 and beginning again at 1, I just go back to 1 and resume from 2 again. So, what I have done just to trace the path here is I have done, 1, 2, 4, 5, 6, 1, 2, 3, 2, 4, 6, 1, 7 that is this one long test path that is here.

The second one reverses this order of visiting 4, 5, 6 and 4, 6. It does 4, 6 first and then 4, 5, 6. So, four answers for edge coverage do 1, 7 separately and then the rest of these parts visit them in any order that you like, but you have to be able to do this large strongly connected component twice, once going through this path and once going through this path and there is a choice here. Similarly to once if you exercise this then the next time you have to be able to exercise this that is the only way I can cover all the edges in the graph. So, that is the test path for edge coverage.

(Refer Slide Time: 16:02)





Question 7

Question 7: Consider the simple path [3, 2, 4, 5, 6] and the test path [1, 2, 3, 2, 4, 6, 1, 2, 4, 5, 6, 1, 7]. Does the test path tour the simple path directly or with a side trip? If it tours with a side trip, identify the side trip. [1+1 = 2 marks].

Answer:
The test path tours the simple path with a side trip.
The side trip it takes could be written as one of the following:

- Path [3, 2, 4, 6, 1, 2], or
- Path [2, 4, 6, 1].

The last question of this assignment asks you to go back to the graph and consider the simple path 3, 2, 4, 5, 6. So, let us go and see what that simple path looks like. It is this: 3, 2, 4, 5, 6. Please remember simple path is not a test path. So, it can begin an end at any node the only thing to note is that it is a simple path does not have any cycles.

So, suppose you had a specified path coverage requirement of having to write a test path that covers this simple path. So, they have made an attempt, given you some test path. So, the test path very long it has 1, 2, 3, 2, then that is the 4, 6 edge comes back to 1 and that is the 4, 5, 6 edge it comes back to 1 and then to 7. It is this long path the just walks through the entire graph once round and round. So, the question that was asked were two parts does this test path toward the simple path directly or with a side trip? The test path does not toward simply path directly right, if you see the simple path is 3 to 4, 5, 6 it is split into 2 parts a 3 and 2 come here, 4 and 5 and 6 come here.

So, it does not toward simple path directly. If it was the simple path directly the simple path will be a contiguous sub path of the test path. Because it is not there it does not do other simply path directly. So, the test path in fact, tours the simple path with a side trip. What is the side trip? Side trip is this bit that is left in between. So, 4, 6, 1, 2, so, you could write it like this 2, 4, 6, 1 or you could include 3 and these 2 and make it 3, 4, 6, 1, 2. So, this test path tours this is given simply path with the side trip and the side trip looks like this if you are written either of these it would be acceptable.

So, I hope this small exercise helped you to understand how to solve simple assignments that we would be giving every week. I will try to upload a couple of more videos as we move along for other assignments that will help you to solve these assignments.

(Refer Slide Time: 18:08)



For now, to get to know more about the graph structural coverage criteria I encourage you to look at the web app that comes as a part of this course book. It is a very nice app. It will help you to learn how to represent graphs as adjacency lists, and how to feed them and how to do various coverage criteria. Please do try it out hope it be useful to you.

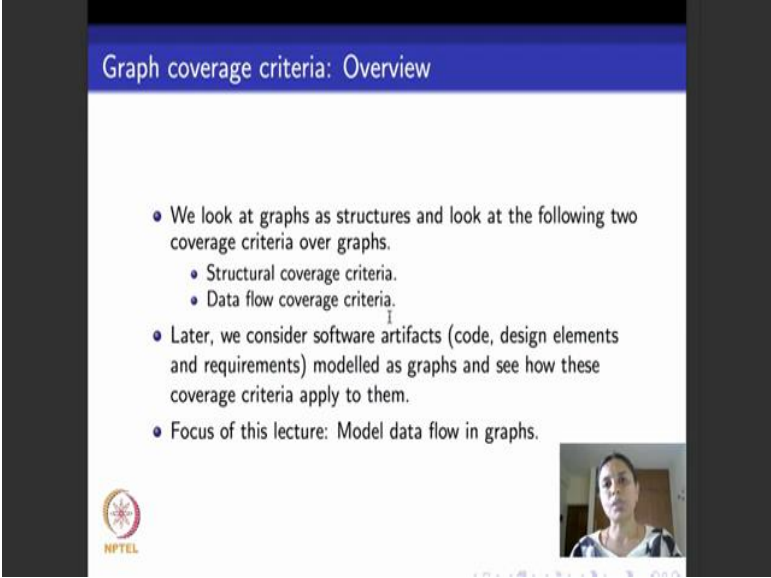
Thank you.

Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 11
Data flow in graphs

Hello again, from today's lecture onward we'll continue with graphs, but we will begin to look at not only control flow and also about discussing about modeling data and how to write test cases and define test paths that deal with handling about how data flows.

(Refer Slide Time: 00:30)



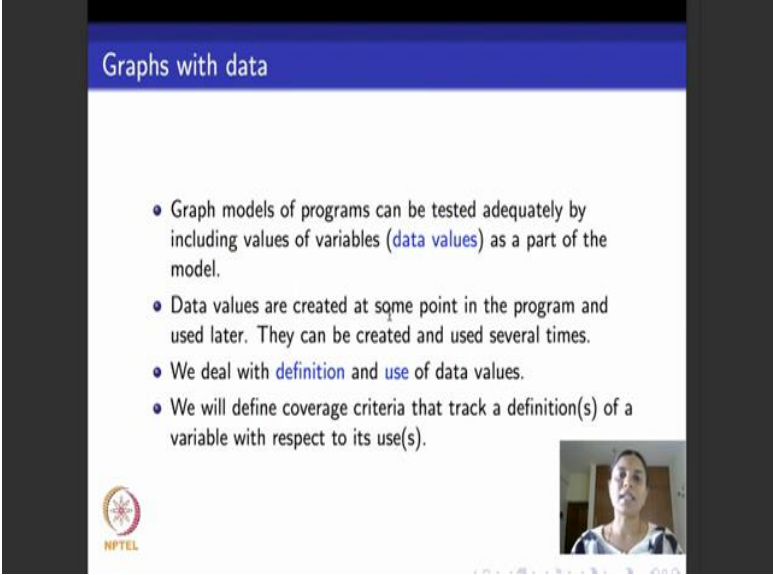
The slide is titled "Graph coverage criteria: Overview" in a blue header. It contains four bullet points: "We look at graphs as structures and look at the following two coverage criteria over graphs." (with sub-bullets "Structural coverage criteria." and "Data flow coverage criteria."), "Later, we consider software artifacts (code, design elements and requirements) modelled as graphs and see how these coverage criteria apply to them.", and "Focus of this lecture: Model data flow in graphs." In the bottom right corner, there is a small video inset showing a woman speaking. The NPTEL logo is in the bottom left corner.

Just to recap where we are in the course. We are looking at a test case generation algorithms and test requirement definitions, based on graph coverage criteria. In the previous week, we saw structural graph coverage criteria which is just coverage criteria based on the structure of the graph, vertices, edges, paths and so on. This week I will begin with dealing with a second part which is data flow coverage criteria and then after we do these two, very soon we will look at various software artifacts core design requirements, see how to model them as graph and recap all these structural coverage criteria and data coverage criteria that we saw and see how to use them to actually test these software artifacts.

So, the focus of today's lecture and for a couple of lectures more to go would be related to data flow coverage criteria. What we will be seeing today is what is data in a software

artifact mainly programs as far as this is concerned and how to model data in graphs. In the next module I will tell you about once we have models augmented with data, how to define coverage criteria and what is the state of the art when it comes to defining test paths that achieve these coverage criteria.

(Refer Slide Time: 01:51)



The slide is titled "Graphs with data" in a blue header. It contains a list of four bullet points:

- Graph models of programs can be tested adequately by including values of variables (**data values**) as a part of the model.
- Data values are created at some point in the program and used later. They can be created and used several times.
- We deal with **definition** and **use** of data values.
- We will define coverage criteria that track a definition(s) of a variable with respect to its use(s).

In the bottom left corner, there is a small circular logo with the text "NPTEL" below it. In the bottom right corner, there is a small video inset showing a person speaking.

So, what is data? As far as programs are concerned, as far as many software artifacts are concerned there is only one kind of data available. Data is available as variables. Variables could have types: variables could have simple types variables could be complex types like arrays several other user defined types and so on, but whatever it is all the data that typical program deals with is basically represented as variables. What are these variables used for in the program? They are used because they represents some kind of information or data, but there are basically 2 kinds of basic ways in which we subject variables to the program. One is to say that a variable gets created at some point in the program typically when it is declared, when it is declared and initialized. And at some point later in the program a variable gets used. So, it could be the case that certain declarations get missed or certain use is get missed.

Suppose the variable that is not declared suddenly gets used in a program then; obviously, the program will not compile. It will be a compiler error, compiler will say that there is a variable that is not well declared in the program, but think of the other way round right a variable is declared in the program, but never used at all. Maybe the

program was large and this programmer meant to use this variable. So, he declared it, but did not really use it because it was wrongly declared. We really do not want such things to happen.

So, we want to be able to track a variable from the place where it is defined to all the places where it is used. And we would not check whether it is used at all or not. So, what we will deal with throughout this lecture would be how to define data places where variables are used, and what are the places where they are actually put to use. And in the next lecture we look at coverage criteria that will track the definition of a variable with reference to it is use.

(Refer Slide Time: 03:53)

Definition and use of values

- A **definition (def)** is a location where a value of a variable is stored into memory.
 - It could be through an input, an assignment statement etc.
- A **use** is a location where a value of a variable is accessed.
 - It could be assigned to another variable, be a part of an if, while or other conditions etc.
- Values are *carried* from their defs to uses. We call these **du-pairs** or def-use pairs.
- A du-pair is a pair of locations (l_i, l_j) such that a variable v is defined at l_i and used at l_j .

NPTEL

So, what is the definition of a variable? A definition of a variable is a location in the program where the value of the variable is stored into memory. That location could be an input statement right, it could be an assignment statement, it could be parameter passing from one procedure to another, could be any kind of statement. We will call statement abstractly as locations so that we can smoothly switch between using statement is a location and when we look at graphs nodes for as will become location. So, just to recap what is definition of a variable with use the word def for short; it is any location in the program where a variable is stored into memory it is value is stored into memory, assignment statements, input statements, parameter passing, procedure calls through parameter passing all these could be places where a variables value is defined.

Now, a defined variable has to be put to use. So, what is a use? A use is a location in a program where a variable's value that is stored is accessed by the statement of a program, by an assignment statement where and it could be assigned to another variable, it is accessed because it is being checked as a part of a condition that comes along with an if statement or it is checked, it is used as a part of a condition that comes along with the loop like for loop or while loop. So, whenever a value of a variable is accessed you say that that is a use corresponding to a variable. So, variable is defined and then in one or more ways and then a variable is used in one or more ways and typically a program carries the value of each variable from its definition to its use right. So, these pairs of definitions and uses of a variable are what are known as du-pairs or def use pair.

So, what is a du-pair a du-pair is a pair of locations say (l_i, l_j) such that a particular variable v is defined at l_i and used at l_j . Please note that even though I've not mentioned here v is an explicit parameter toward du-pair per variable they could be several du-pairs right, per pair of location there could be several variables that are defined and used at any point of time given a variable you are looking at what it is a du pair and given a pair of location, you are looking at all the set of variables that are involved in that location definition. And use of that location we may or may not mention it explicitly in this lecture, but there is always a parameter or an attribute to every definition or a use or a du pair which is always a variable, because if variables are not there what do we define and what do we use.

So, what did we learn till now? We learned what a definition of a variable is. It is basically a place where if variable values first time written into memory: could be an input statement could be a declaration could be an initialization could be parameter passing several things what is a use a use of a variable is a statement or a location where a value of the variable is accessed it is use could be because it comes on the right hand side of an assignment statement or it is used as a part of a predicate that comes for checking for an if or for a loop condition. What is a du pair a du-pair is a pair of locations (l_i, l_j) such as the variable is defined at l_i and used at l_j ?

(Refer Slide Time: 07:15)

Data in graphs

- Let V be the set of variables that are associated with the program artifact being modelled as a graph.
- The subset of V that each node n (edge e) defines is called $def(n)$ ($def(e)$).
 - Typically, graphs from programs don't have defs on edges.
 - Designs modeled as finite state machines have defs as side effects or actions on edges.
- The subset of V that each node n (edge e) uses is called as $use(n)$ ($use(e)$).

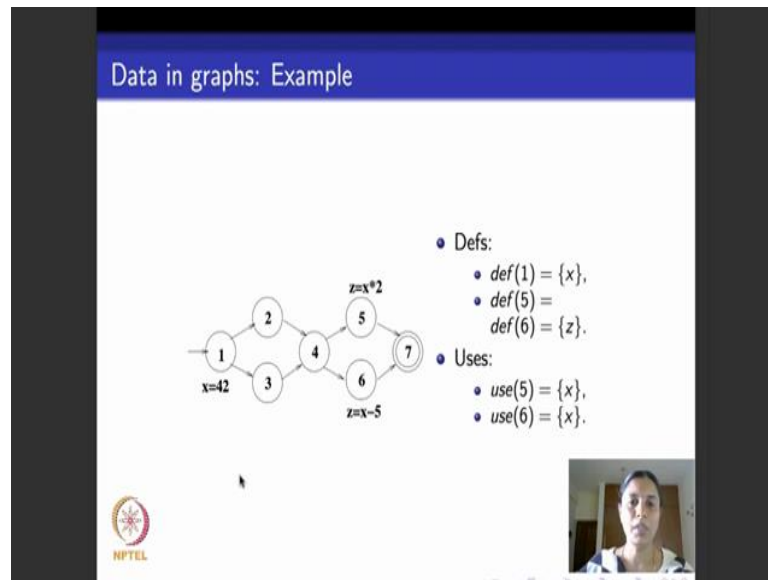
NPTEL

So, consider program that are let us say modeled as a graph. We will look at graph or data flow testing with reference to graphs in this module. So, we consider a program that is modeled as a graph, and let v be the set of variables that are associated in the program right. I want to know how these variables now come inside this graph right. So, what I do is I take the graph then graph has nodes and edges and per node and per edge, wherever it is applicable, I will say which are the variables that are defined in that node called as def of n for a node n . And for an edge e I will say which are the variables that are defined in the edge e .

Typically graphs that correspond to programs will not have definitions on edges. I mean will not have definitions on edges, as I told you. When is a definition, the definition is when a variable is written into memory. So, it is like having an assignment statement is a part of an edge, it is rare, it is rare or almost impossible in a graph that models a program. But for graphs that come as design models let us say graphs that come as finite state machines modeling designs, it is possible to have definitions on edges. Definitions could be thought of as actions that change the value of a variable or side effects related to an edge right. The subset of the set of variables that each node or edge uses is called use of n or use of e .

So, per node and per edge in the graph, we talk about what are the subsets of variables that it defines and what are the subsets of variables that it uses.

(Refer Slide Time: 08:47)

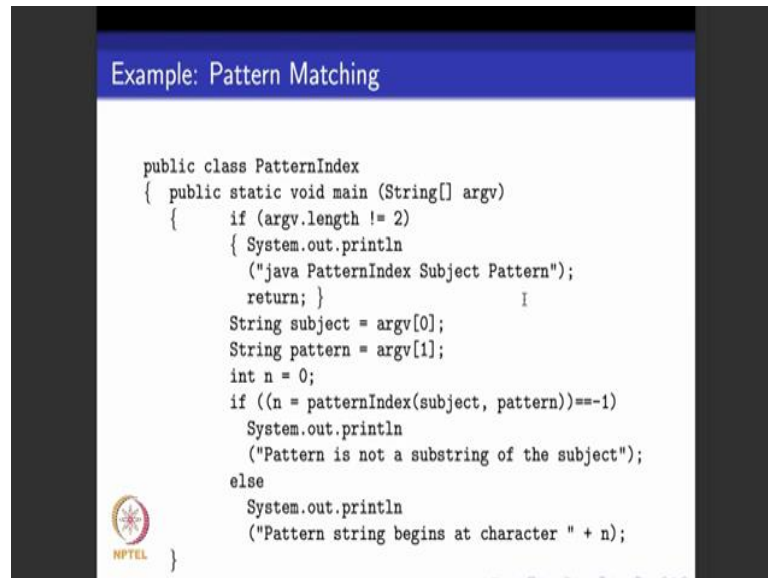


So, here is a small example. Here is a graph that has 7 nodes, initial node is one final node is 7. Not all nodes have defs and uses only few nodes have. What I have a done here? I said at the node 1 there is a statement which says x is equal to 42, at node 5 there is another statement which says z is equal to x into 2, at node 6 there is another statement which says z is equal to x minus 5. So, you can think of this statement x is equal to 42 that comes at the initial node 1, as defining the value of x at node one right. So, we say definition one def of one is this singleton set $\{x\}$. 2 and 3 and 4 nothing happens, I move on, I look at 5. For statements at nodes 5 and 6 have expressions that calculate the value of z based on the value of x right. So, at 5 we have z is equal to x into 2, at 6 we have z is equal to x into 5.

So, nodes, these 2 nodes can be thought of as having statements that define z. So, we say definition of 5 and definition of 6 both are the singleton set $\{z\}$. Now how is z defined at nodes 5 and 6 z is defined using an expression that involves a constant and it involves the variable x. Similarly, here z is defined using another expression which involves this constant 5 and a variable x. So, we said x is used at nodes 5 and 6. So, is it clear? So, the first occurrence where the value of x is initialized or set to some value or declared or read from input is it is definition. In this case the def, x is defined at node one and whenever there is a statement that puts this defined value into use right like here at nodes 5 and 6, we say x is used in those places. In addition to that at nodes 5 and 6 the value z

is assigned an expression involving a x, and so we say x is defined at nodes 5 and 6. We look at a more detailed example to understand this clearly.

(Refer Slide Time: 11:05)



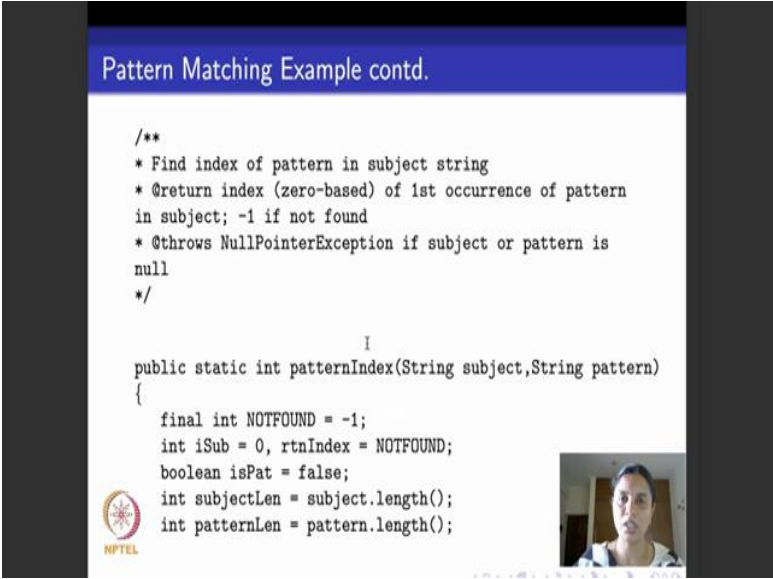
So, here is an example that deals with the Java code corresponding to a pattern matching example. Some version of pattern matching or the other if you look at you will find it in several papers and books related to testing. It is one of the popular examples that we always deal with in testing because mainly because it is got lot of rich control flow structure. So, what I have done here is I have put a Java code for pattern matching that was taken from this text book, Introduction to Software Testing by Ammann and Offutt. What I have done is to make it readable, I have spread across this Java code over 3 slides. So, we first go through the program line by line, well understand what this program does, then well go ahead and draw the control flow graph of this program, and look at how the data or the variables that come in this program can decorate the control flow graph, where is it defined where is it used right. We use this example to understand the defs and uses of all the variables on the control flow graph corresponding to this code.

So, this is probably the first full piece of code that you are looking at and we will reuse this example let a few other places as we go down in our lectures. So, what does this code do? This code takes two arguments: a subject and a pattern, both of them are strings. And then it looks searches for a pattern in that subject. Like for example, I could

say subject is a string which says institute, pattern could be something like ins right. If ins as a pattern comes in the subject, then you say that this pattern is found in the subject and you return index at which this pattern begins in the subject. If the pattern is not found in the subject then you return a string called -1. So, that is what it says. So, it says that you return -1. If the code returns -1 then your output saying pattern is not a substring of the subject. If the code returns any other character, then you say pattern does occur in the subject as a substring. And it begins at this particular index, yeah one more thing to note is this we look at a pattern as a contiguous substring we do not look at it as bits and pieces.

So, this is the first slide containing the first half of the code, which does on the initializations in the main prints .

(Refer Slide Time: 13:30)



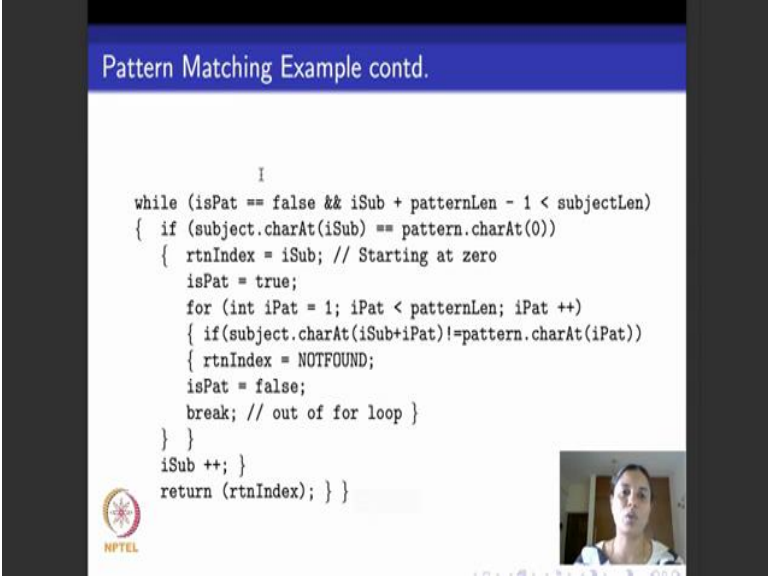
Pattern Matching Example contd.

```
/**
 * Find index of pattern in subject string
 * @return index (zero-based) of 1st occurrence of pattern
 * in subject; -1 if not found
 * @throws NullPointerException if subject or pattern is
 * null
 */
public static int patternIndex(String subject,String pattern)
{
    final int NOTFOUND = -1;
    int iSub = 0, rtnIndex = NOTFOUND;
    boolean isPat = false;
    int subjectLen = subject.length();
    int patternLen = pattern.length();
```

The actual code is given in continued across 2 more slides. So, here is the code that is continued from the first slide into the second slide. So, as I told you what is the goal of this code ? It is to find the index of the pattern, if it occurs in the subject and it will return the index 0 based in the sense that we count indices starting from 0, the first character of subject is index is 0, right. If the first occurrence of pattern in the subject we return -1 if it is not found if subject or pattern is empty then we threw a null pointer exception right. How does this pattern index work ? As I told you takes 2 arguments a subject and a pattern, and it does all these kinds of initialization?

So, it uses these variables not found, it initializes it to -1 in integer variable. It uses 2 other integer variables, isub you can read it as index that moves across the subject it initializes the index of the subject to 0. And then it sets the return index to be -1 which is not found. Remember when the return index continues to be -1 if you go back to the previous slide I return saying pattern is not a substring of the subject. And then it uses a few Boolean variables, ispat standing for 'is pattern', initializes it to false and then it calculates the length of the subject and the length of the pattern. Why do we need the length of the subject and length of the pattern ? Because we want to use a for loop to move across the length of the subject from left to right and another for loop to move across the length of the pattern from left to right?

(Refer Slide Time: 15:08)



Pattern Matching Example contd.

```
I
while (isPat == false && iSub + patternLen - 1 < subjectLen)
{
    if (subject.charAt(iSub) == pattern.charAt(0))
    {
        rtnIndex = iSub; // Starting at zero
        isPat = true;
        for (int iPat = 1; iPat < patternLen; iPat++)
        {
            if (subject.charAt(iSub+iPat) != pattern.charAt(iPat))
            {
                rtnIndex = NOTFOUND;
                isPat = false;
                break; // out of for loop
            }
        }
        iSub++;
    }
    return (rtnIndex);
}
```

So, here is the main part of the code. There is a while loop inside this there is a condition there is a for loop, and then there is an if loop. Why am I talking about this while if for if, is to tell you that this code has very rich control flow structure. The control flow graph of this code will involve some amount of branching which is a very which makes it a very interesting case for testing and here there are lots of nested loops one inside the other. So, the control flow structure it becomes even richer and so it is an interesting example to look at for testing. Now going back and understanding what the code is. So, it says as long as it 'is pattern' turns out to be false and pattern index is within the subject of the length of the pattern is within the subject index then what do I do in this if loop, I start matching character by character.

So, I say wherever I am in the subject which is this index of subject isub, does it match the first character of the pattern? First character of the pattern is present at index 0. If it is then, you say that the return index could be this value. Let us say if you take the earlier example that we had right, let us say a subject was the string institute and the pattern was ins, so, right up front at the index 0 itself, the subject and the pattern match, i and i match right. So, you set the index where it matches to be the return index and then you set this Boolean flag is pattern found which is ispat to be true right.

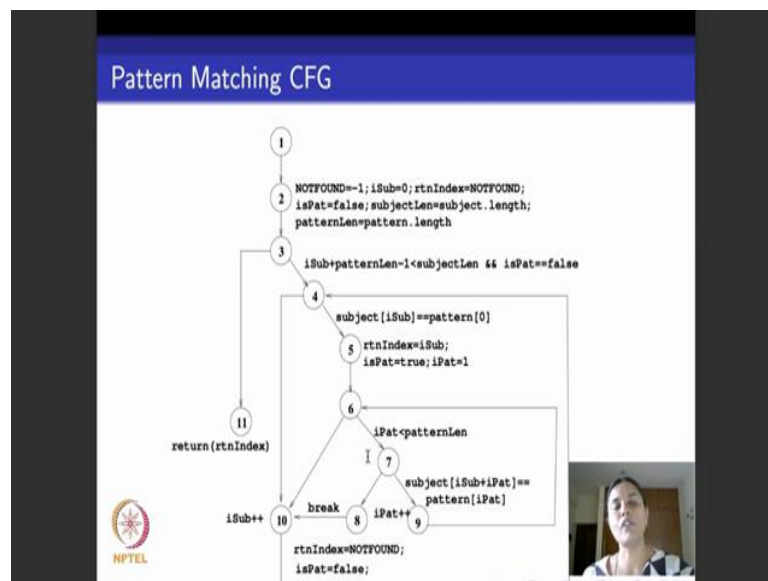
Now what you do you enter a for loop, keep incrementing the position that you look at in the pattern as long as you go till the end of the pattern and see if every character of the pattern, one at a time, matches the corresponding character in the subject that is what this if statement does. If there is a mismatch at some point in time it says he sets the return index to not found and makes is pattern false and comes out of the for loop right. If every character of the pattern continues to match the subject it keeps incrementing the index in the subject successfully finishes the for loop and it will return this value here that it is set as the return index. So, is it clear what the code does?

So, I just to quickly recap we are looking at this pattern matching code. It takes 2 arguments, a subject and a pattern and it uses a few Boolean variables to search for the pattern in the subject. What it does is initially it assumes that the pattern is not found in the subject. So, it says 'is pattern' is false, and then here it checks with the length of the pattern is well within the length of the subject and then it says now let us look at the first character of the pattern which is at index 0 along with wherever I am in the subject which is given by this index isub, if they match if this return statements gives true, then you say I may have found the pattern beginning at this index.

So, you set that to be the return index. And then you say I have found the pattern set you to true, then you enter a for loop where you continuously walk down the pattern and check if every character in the pattern that you encounter, in sequence, matches the subject from the index in the same sequence. If there is a mismatch at any point in time then you reset a return index to not found say pattern is not found come out of the for loop. But if we successfully finish executing this for loop then you increment the subject and then you can return this value of the index where the pattern was found in the subject.

So, now our first goal is to draw the control flow graph corresponding to this program right. So, as I told you for control flow graph this program is spread across these 3 slides, but I am ignoring this part of the code, and this part of the code which is just commented out. I begin to draw control flow graph mainly from here, from this main pattern index method. So, we see this pattern index method it has a whole series of assignment statements, it has a loop, a branch, another loop another branch.

(Refer Slide Time: 19:15)



So, here is how the control flow graph of that main method looks like. This node one is a dummy node which represents this statement. It says subject and pattern are passed to this node. There is no concrete statement representing this so I have left it blank. Node 2 stands for all these initializations that you do in the beginning of the program right all these statements these 6 statements that are there. You could put it in different nodes, but typically when we draw control flow graph, when we have a sequence of assignment statements without any control statement in between just assignment statements one after the other, it is a standard practice to collapse them all into one node and put it as one node and then augment that node with all these statements. That is what I have done because there is not much point in calling each of them as separate node, we really do not find any use for it when it comes to testing.

So, we collapse it all and put it into one node. So, this single node represents all these assignment statements. After that what do we do with that code we get into this while

loop right. So, that is this while loop at node 3, I say this is the condition that I check in the while loop you can see that is the condition I check. This condition could be true this condition could be false. Just for increasing the readability I have written it only as a label for this node the reverse of this condition the negation of this predicate is true here. I have not mentioned it because the graph was becoming the figure was becoming too cluttered. So, I have just left it at that right. So, read the label of this edge as negation of this.

So, I am entering my while loop here. As per the code the next is an if statement which check for another condition. So, that I am entering my if statement here, if statement returns true, this is if statement returning false. After my if statement returns 2, I do two assignments I set, what does it mean for the if statement to return true I have found the first match. First character with the pattern has found a match in the subject. So, I do these 2 assignments if you remember right I set the index and say I have found the first match by setting the flag to be true. So, that is this node 5.

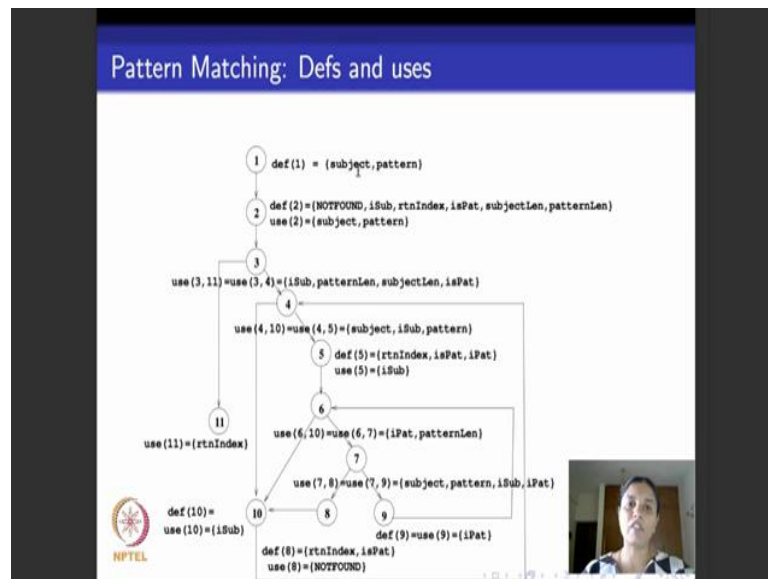
After that I enter, sorry, after that I enter this for loop, that is this. This is the statement corresponding to the for loop being true. And inside the loop I keep checking if the pattern index matches the subject index. At node 9, I increment the pattern index go back and repeat this for loop here from 6 to 9 and when the condition fails I come out with a for loop. I break out of the for loop and I am here that is what this represents and then this is the thing that they use to go back for the main while loop. This is when I exit the main while loop, I do this final statement which is return index. Is it clear? I am sorry I have to go back and forth across the slides because there is no way I could have put the whole a program and the control flow graph into one slide.

So, just to summarize we have looked at a pattern matching program that I had presented to you across 3 slides which were this, this and this. It is a program that has rich control flow structure looks if pattern given as a string comes with a substring of a subject. What we did here was to take the main method in the program and draw its control flow graph.

So, I will repeat this exercise of how control flow graph will look like for various program constructs, but hopefully this example will help you to understand it also. Now we go back to the main goal of this lecture which is to understand data right, data

definition and data use. What is data for this particular program? Data is all these variables: NOTFOUND, isub, rtnIndex, subject, pattern, ispat. So, many variables are there right. So, what do I do, I take the same control flow graph instead of annotating it with statements I say which are the variables that are defined and which are the variables that are used at various nodes in the graph and various edges in the graph.

(Refer Slide Time: 23:46)



So, that is this graph that I have drawn here. It is a same control flow graph instead of depicting statements it now depicts definitions and uses. So, if you look at node number one, it says the definition of one that is at node number one, which are the two variables that are defined? The Two variables that are defined at one are subject and pattern. You might wonder this here it is blank and while have I suddenly put def(1) as subject and pattern as I told you when you explain the CFG read this node one as these two patterns subject and pattern being passed on as parameters to this method. Because this passed as parameters I am not depicted here, but because it is passed as parameters it is defined here.

Now, what is the def (2)? If you see there are, so many assignment statements here which is not found as -1, isub is 0, rtnIndex is NOTFOUND, ispat is false and so on right. So, I say definition of 2 is all the variables that have been initialized here, which is NOTFOUND, isub, rtnIndex, ispat, subject length, pattern length. Now 2 also has two uses which is subject and pattern. Why is it use of subject and pattern I use the variable

subject and pattern that was passed as parameters at node 1 and compute their length here subject length and pattern length and assign it to subjectLen and patternLen. So, I have used these 2 variables subjectLen and patternLen. So, that is why I have put use of 2 as subject and pattern right. Is one more settle point do not worry about it. If you do not understand it if you see I have initialized not found to be minus 1. So, I have to def(2) as not found and I have also initialize a return index to be not found.

So, strictly speaking I should put use(2) as not found. So, you could go ahead and do that, but at the little later well see that we typically do not consider what are call local uses of a variable. Because I have collapsed all these 5, 6 assignment statements into one node in the control flow graph, I say this variable is defined and used within one node. So, it is like a local use of a variable. We typically do not capture local uses of variables, but if you are not comfortable with this if you want to see use(2) it is subject pattern and not found you can go ahead and do it there is no harm in it right like this I move on.

Now, I can define defs and uses for edges also remember well saw definition and uses for vertices and definition and uses for edges. So, which is this, if a predicate that labels this edge. So, it uses thus a variables isub, patternLen, subjectLen and ispat. So, that is what I have written here. The use of this edge is this predicate i is this set isub pattern length subject length ispat. If you remember I told you that this corresponds to a branch, this is the branch where I enter the while loop. So, this predicate is true. This is the branch where I exit the while loop where the predicate becomes false.

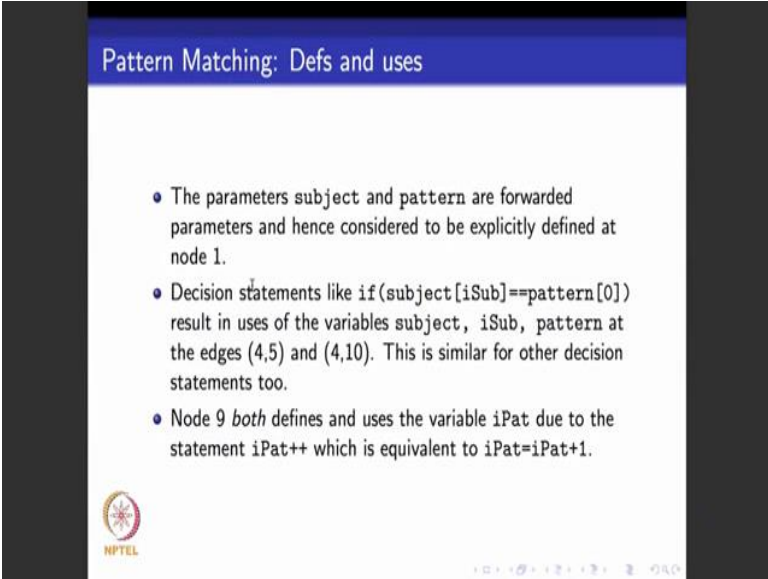
Just for readability I didnt write the condition here, but the same set of variables with this condition negated comes here also. So, I say use of this edge (3, 11) and this edge (3, 4) is the same set right. Because the both the predicates one the positive version and one the negated version both will use the same set of variables. Similarly, this is the if statement. So, use of this edge (4, 10) and this edge (4, 5) is this set subject, isub, pattern and then at statement 5, I have these 2 assignment statements. So, I say rtnIndex, ispat and ipat are defined at the statement. Please note that ipat is defined once here also sorry, ispat is defined once here also at 2 and it is defined at 5.

So, if you go back and look at the code it is initialized at 2 and its value is reset to another value at 5. So, both we call them as definitions because it is again written, rewritten in memory. So, definition of node 5 is this set. Similarly use on node 5 is this

isub, is this isub, is it clear. So, if I move on like this. Use of this edge (6, 10) and this edge (6, 7) are all the variables that come in the predicate corresponding to the for loop, use of this edge (7, 8) and the edge (7, 9) are all the predicates that come in the if loop. And then what do I do at 9 at 9? I do this, I increment the index of searching in the pattern by using ipat+. So, what is ipat++, that reads as ipat is equal to ipat plus 1. So, if I read it that way I am using ipat both in the left hand side and in the right hand side. So, it is both defined and used at variable 9.

So, that is where def of 9 is the same as use of 9 and that is the single concept ipat right I go on like this finish right. So, what do I do? Just to recap, we have a program like this. We define the control flow graph corresponding to the program, from the control flow graph I can take the basic control flow structure and instead of augmenting with statements I augment it with defs and uses of all the variables. So, is it clear how to do it? So, before we move on I will go back to the slide where we define defs and uses. So, what is a def, is a place wherever the value of the variable is stored into memory. What is a use, a place where value of the variable is accessed in a memory right? Typically, we take a graph and for every node in the graph, we call what is def of n and for every edge in the graph we say what is use of n and for every edge in the graph we say what is use of e and def of e.

(Refer Slide Time: 30:12)



Pattern Matching: Defs and uses

- The parameters `subject` and `pattern` are forwarded parameters and hence considered to be explicitly defined at node 1.
- Decision statements like `if(subject[iSub]==pattern[0])` result in uses of the variables `subject`, `iSub`, `pattern` at the edges (4,5) and (4,10). This is similar for other decision statements too.
- Node 9 *both* defines and uses the variable `iPat` due to the statement `iPat++` which is equivalent to `iPat=iPat+1`.

NPTEL

So, this I think I explained this slide already what I said was at node one parameters subjected pattern of a forwarded parameter. So, they consider to be explicitly defined at node one. And I also explained the how these uses occur in decision statements that is what this line item says. And the node 9 has a statement which is like `ispat++`. Suppose I read it as `ispat++` as `ispat` plus 1 then it can be thought of as both defining and using this variable right. So, that is why I write def of 9 is the same as use of 9 is the same as `ispat`.

(Refer Slide Time: 30:50)

Data defs and uses: Other definitions

In testing literature, there are two notions of uses available.

- If v is used in a computational or output statement, the use is referred to as a **computation use** (or **c-use**), and the pair is denoted as $dcu(l_i, l_j, v)$, where v is defined at l_i and used at l_j .
- If v is used in a conditional statement, its use is called as a **predicate use** (or **p-use**).
- For conditional use, two def-use pairs appear:
 - $dpu(l_i, (l_j, l_r), v)$ and $dpu(l_i, (l_r, l_j), v)$, where v is defined at l_i , used at l_j , but has two opposite flow directions (l_j, l_r) and (l_r, l_j) .
 - The former denotes the true edge of the conditional statement in which v is used; the latter the false edge.
- We will not distinguish between the above two uses in this course.

NPTL

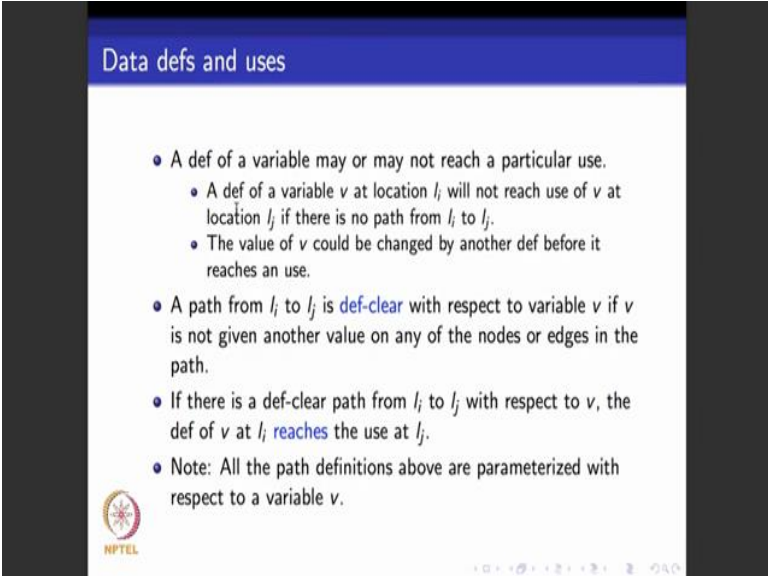
Before we move on, traditionally in literature there is a paper by Wyeuker and a student by name Sarah, which is well quoted for data flow graphs. They distinguish between two different kinds of uses. So, they say when a use occurs as a part of an assignment statement or an output statement that is what is called a computation use or a c-use. And a use could occur as a part of a condition statement right like we saw here this use is a part of a definition at node 2. Whereas, this use at this edge is a part of a predicate or a condition that occurs as a part of the if statement.

So, they distinguish between these 2 uses, one is a computation use or use that occurs as a part of a printf or as an assignment statement is what is called a computation use. A use that occurs as a part of a decision statement is what is called a predicate use. So, instead of talking about du-pairs definition use pairs they talk about definition computation use d c u pairs or definition predicate use d p u pairs. So, when I talk about a definition computation use d c u pair I say a variable v is defined at l_i and is being subject to

computation use at l_j that is it is used as a part of an assignment statement or as a printf statement at l_j . When I talk about d p u I say the variable v is defined at statement l_i it is used in the predicate l_j and l_t says that this predicate at location l_j has evaluated to true and here l_f says the predicate at a location j has evaluated to false.

So, when I use it on as a part of a predicate which could be as a part of an if statement or a while statement, the single purpose of that use is to make the predicate evaluate to true or evaluate to false. So, and based on whether it evaluates to true or false, the paths that it takes in the control flow graph could be different. So, this two paths are distinguished. This is just for your information for our purposes of these lectures and this course we will not distinguish between computation use and predicate use. We will just say all of them are uses. That is what I have done in this control flow graph right. If you have to be the strictly correct you could call this is predicate use you could call this is computation use, but for the purposes of this course and this lecture everything would just be use.

(Refer Slide Time: 33:32)



Data defs and uses

- A def of a variable may or may not reach a particular use.
 - A def of a variable v at location l_i will not reach use of v at location l_j if there is no path from l_i to l_j .
 - The value of v could be changed by another def before it reaches an use.
- A path from l_i to l_j is **def-clear** with respect to variable v if v is not given another value on any of the nodes or edges in the path.
- If there is a def-clear path from l_i to l_j with respect to v , the def of v at l_i **reaches** the use at l_j .
- Note: All the path definitions above are parameterized with respect to a variable v .

NPTL

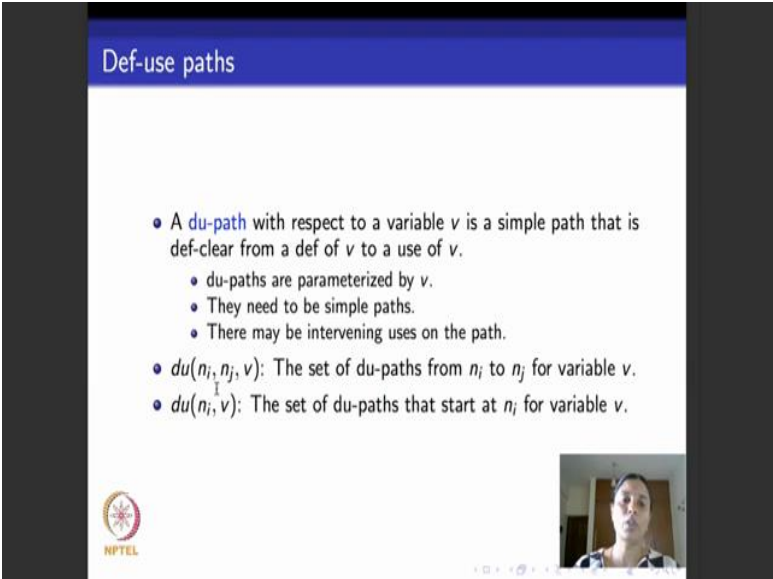
What will be the concerns about definitions and uses? As I told you in the beginning of this lecture a definition may or may not reach a particular use. Why will it not reach a particular use? It could happen, it could have happened that the clear a variable was defined at a particular location and used at some other location l_j . And then there was no path from l_i to l_j , l_j could be a part of a dead code in the program or it could be the case that the value was defined and before it was used it was defined again. It was initialized

and before it was put to use again maybe it was defined by some other statement once again right. So, such things can happen.

So, we talk about several kinds of paths in the control flow graph. So, we say a path from a location l_i to another location l_j is def-clear with respect to a variable v if v is not given any other value on any of the nodes or edges in the path. So, I take a control flow graph augmented with definitions and uses take a location l_j , l_i take another location l_j v is defined at l_i , a variable v is defined at l_i .

Now, there is a path from l_i all the way to l_j which could be another node or an edge. I walk along the path. In between in the path if the variable v is not defined once again then you say such a path is def-clear. If there is a def-clear path from l_i to l_j with reference to a variable v , we say that the definition of v at l_i reaches the use at l_j . As I told you whenever we talk about definitions and use this always implicitly a parameter of a variable that is involved.

(Refer Slide Time: 35:17)



Def-use paths

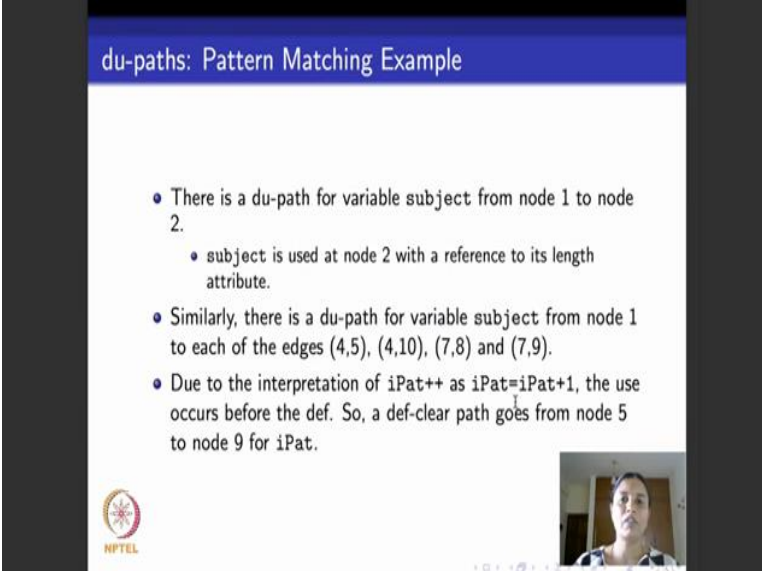
- A **du-path** with respect to a variable v is a simple path that is def-clear from a def of v to a use of v .
 - du-paths are parameterized by v .
 - They need to be simple paths.
 - There may be intervening uses on the path.
- $du(n_i, n_j, v)$: The set of du-paths from n_i to n_j for variable v .
- $du(n_i, v)$: The set of du-paths that start at n_i for variable v .

So, now, we talk about what is called a def use path or a du-path. What is a du-path? A du-path with respect to a variable v is a simple path that is def clear from a definition of v to a use of v right? So, what it says in simple terms is that take a node or an edge, where a variable is defined in our case it will be a node a node where a variable is defined keep going down and take it is first use right. In between assume that the variable is not defined and the only other condition that we insist is that this path be

simple, there are no cycles in this path. If such is the case then we say that I have found a path from a definition of a variable to a use of a variable and I abbreviate it and call it as a du-path right. So, du-paths have to be simple paths and they have to go from a definition to a use. In between there could be intervening uses, but no definitions.

So, here is a notation for that, you say $du(n_i, n_j, v)$ means it is the set of all definition use path from location n_i to location n_j for the variable v . What is (du, n_i, v) ? It is a set of all du paths that start at n_i for a variable v .

(Refer Slide Time: 36:38)



du-paths: Pattern Matching Example

- There is a du-path for variable subject from node 1 to node 2.
 - subject is used at node 2 with a reference to its length attribute.
- Similarly, there is a du-path for variable subject from node 1 to each of the edges (4,5), (4,10), (7,8) and (7,9).
- Due to the interpretation of `iPat++` as `iPat=iPat+1`, the use occurs before the def. So, a def-clear path goes from node 5 to node 9 for `iPat`.

NPTel

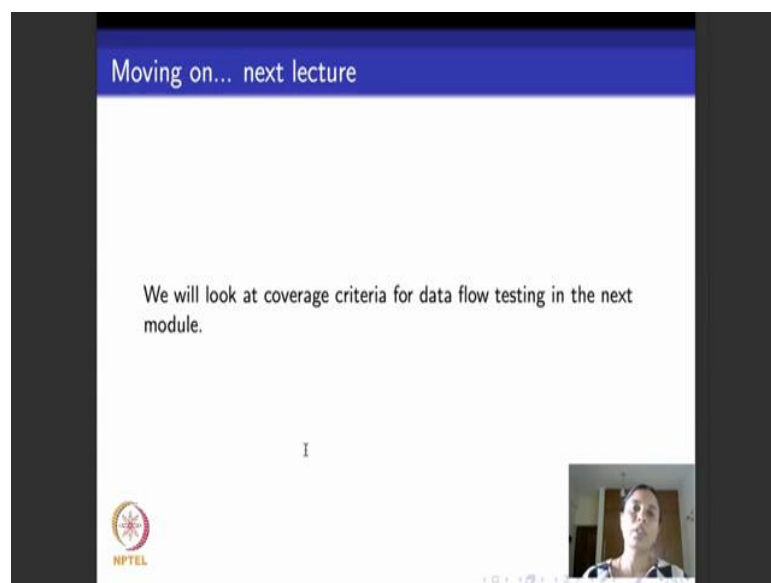
If we go back and trace out what are the du paths. So, if you see there is a small du path here. There is a definition of subject here and then there is a use of subject here. So, I can say if there is a definition use path for the variable subject. So, there is a du -path. Similarly the term subject is defined here and look at other nodes where it is used. Like for example, if you see it is used in this edge (4, 10) and (4, 5) it is you would come as use of (4, 10) and (4, 5) and in between it is not defined again.

So, this path from 1, 2, 3, 4 to the edge (4, 5) or to the edge (4, 10) is a du -path for the variable subject. So, hopefully it is clear what I am saying. There is a du -path for the variable subject from the node 1 to the edges (4, 5), (4, 10). Similarly, if you go back a there is a du -path for the variable subject from the node 1 to (7, 9) and (7, 8), because it is being used here also, but not defined anywhere in between. Please remember a du -path is a path between a definition and a use such that there are no definitions in between in

the path of that variable. There could be uses in the path of that variable, that is all right. Like for example, if I take the du-paths from 1, node 1 to the edges (7, 8) or (7, 9), there is a use of the variable subject in between we allow that, but we say in between a du-path in intermediate nodes or edges there should not be another definition, a du-path with respect to a variable v is a simple path that is def clear from definition to a use, but there could be intervening uses on the path.

So, another example of a du-path for that pattern matching thing is this `ipat++`, which was if you remember at node number 9, if I read `ipat++` as this statement then I can interpret it as the use occurring before the def because this right hand side needs to be executed before the left hand side. So, I with that interpretation I can say that there is a def clear path that goes from node 5 to node 9 for the variable `ipat`.

(Refer Slide Time: 38:59)



So, that is all I have to tell you for today's lecture. What we did today just to summarize is assuming that we have a model of a software artifact typically a code as a graph, which means as a control flow graph how I actually look at the variables that come in the in code, and decorate the control flow graph with the definitions and uses of a variable. So, once I do I understand when a variable is defined and when a variable is used, it could be defined and used at nodes. It is typically used only at edges not defined at edges, then I talk about a du-path or a def clear path and so on. What we will do in the next lecture is to see how these du-paths can be used to define various coverage criteria

that will help us to understand, when a variable is defined is it being used, is it being not used and so on and so forth. So, we look at data flow coverage criteria based on definitions and uses in the next lecture.

Thank you.

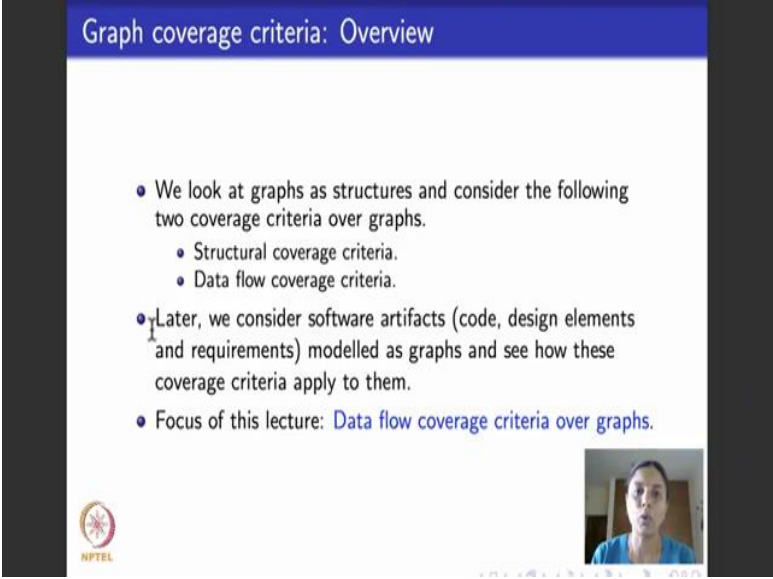
Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 12
Algorithms: Data Flow Graph Coverage Criteria

Hello everyone. Welcome to the next module. The focus of this module is to continue with data flow, if you remember in the last video, we had defined what is data flow in a graph; what was definition of a data; what was use of data and how to track a data from its definition to use? And we saw an example of how this is used.

So, what we will see today is definitions of criteria that are based on data flow: data defs and data uses. And then also tell you briefly about how to define or work on algorithms that will help us to achieve this data flow coverage criteria. Data flow coverage criteria algorithms is a very vast area, early papers came out in the early 80s and this still active research going on. I will not be able to cover all the algorithms that deal with data flow coverage criteria, what will point to you at the end of this lecture would be some links to good reference material you may you could read out more to get to know about algorithms related to data flow.

(Refer Slide Time: 01:12)



Graph coverage criteria: Overview

- We look at graphs as structures and consider the following two coverage criteria over graphs.
 - Structural coverage criteria.
 - Data flow coverage criteria.
- Later, we consider software artifacts (code, design elements and requirements) modelled as graphs and see how these coverage criteria apply to them.
- Focus of this lecture: **Data flow coverage criteria over graphs.**

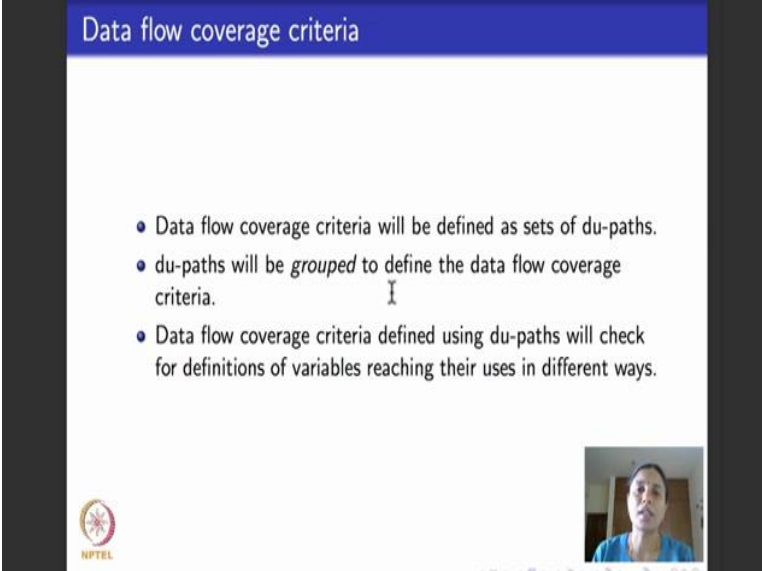
NPTEL

Navigation icons: back, forward, search, etc.

So, just to recap what we did till now we are at the module where graphs are our models that we use to model software artifacts, we saw structural coverage criteria over graphs

then we looked at basic algorithms over graphs algorithms for defining test requirements and test paths for structural coverage criteria. Then I moved on to defining data flow and graphs.

(Refer Slide Time: 01:37)



The slide is titled "Data flow coverage criteria" in a blue header. It contains three bullet points:

- Data flow coverage criteria will be defined as sets of du-paths.
- du-paths will be *grouped* to define the data flow coverage criteria.
- Data flow coverage criteria defined using du-paths will check for definitions of variables reaching their uses in different ways.

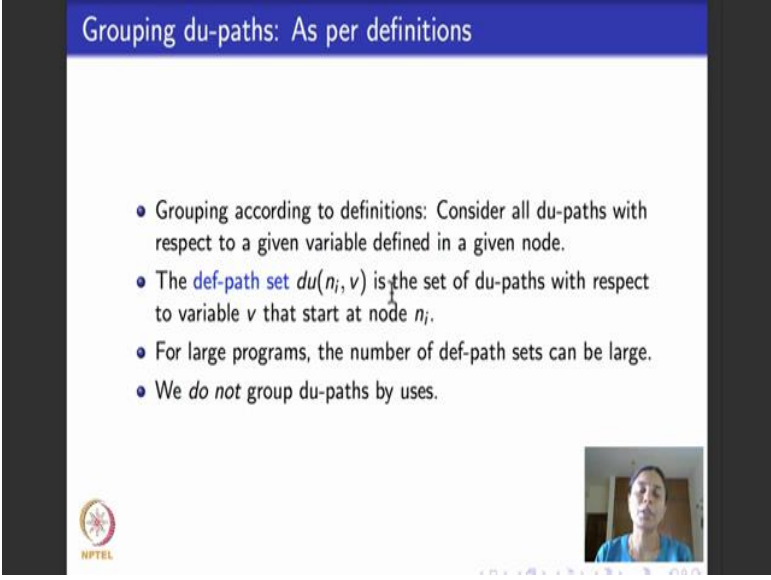
In the bottom right corner, there is a small video inset showing a man speaking. The NPTEL logo is visible in the bottom left corner of the slide.

Today, what we will be seeing in the data flow coverage criteria. How is data flow coverage criteria defined? Data flow coverage criteria is basically defined as a set of du-paths, d for definition, u for use. What is the du-paths? if you remember from the last lecture du-path is a path corresponding to a variable that is given as a parameter to a path that begins at a definition of a variable and goes all the way till the use of the variable, intermediate in this path from it is definition to it is use.

We insist that the variable does not get defined once again. So, the path is definition clear for this variable right. So, what is du-path, it is a path from a definition of a variable to the use of a variable. Such that a every intermediate node on the path there is no further definition of the variables.

What we will do is we will group various kinds of du-paths to be able to define data flow criteria. What these criteria will basically check is they will basically check how a definition of a variable reaches it is use.

(Refer Slide Time: 02:38)



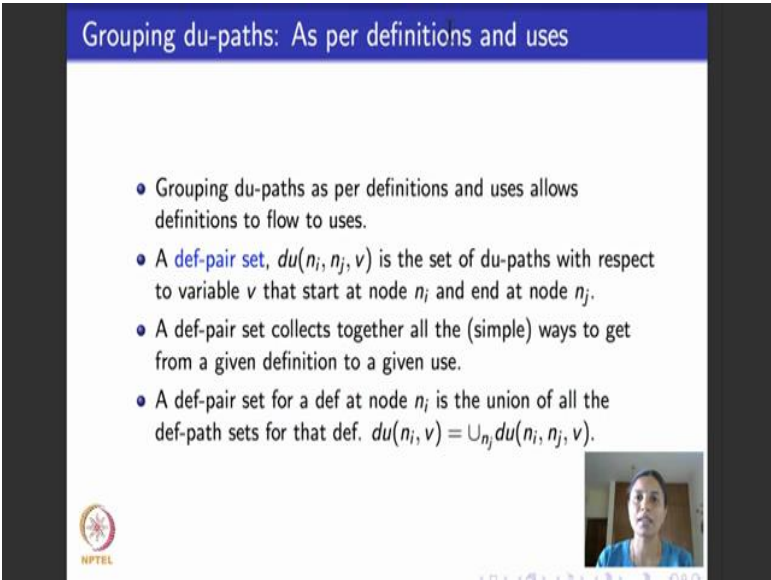
Grouping du-paths: As per definitions

- Grouping according to definitions: Consider all du-paths with respect to a given variable defined in a given node.
- The **def-path set** $du(n_i, v)$ is the set of du-paths with respect to variable v that start at node n_i .
- For large programs, the number of def-path sets can be large.
- We *do not* group du-paths by uses.

NPTEL

However, were going to group du-paths? We are going to group du-paths in 2 different ways the first grouping that we will discuss is as per their definitions, the next grouping that we will discuss is as per their definition and use.

(Refer Slide Time: 02:50)



Grouping du-paths: As per definitions and uses

- Grouping du-paths as per definitions and uses allows definitions to flow to uses.
- A **def-pair set**, $du(n_i, n_j, v)$ is the set of du-paths with respect to variable v that start at node n_i and end at node n_j .
- A def-pair set collects together all the (simple) ways to get from a given definition to a given use.
- A def-pair set for a def at node n_i is the union of all the def-path sets for that def. $du(n_i, v) = \cup_{n_j} du(n_i, n_j, v)$.

NPTEL

So, how are we going to group du-paths as per definition? So, when we group d u path according to a definition we consider du-path with respect to a given variable defined in an given node, and we define it like this. So, v is the variable that whose definition and use we are tracking. It so happens that v u v is designed at this vertex n_i in the graph. So,

what is a def path set called du of the variable v at the node n_i ? It is the set of all d u paths with respect to the variable v that start at the node n_i .

What is the du-def path set? A def path set at a node n_i for a variable v is the set of all du-paths that begin at that node n_i for that variable v . Please note that the number of such def paths for a large program fairly reasonable size program can be very large, but we still have to find them and work on them to be able to define data flow coverage criteria? It is also work noting at the stage that while we group or def du-path with respect to a place where it begins to get defined, we do not really group it with reference to it is uses. So, turns out that in literature grouping it with reference to it is uses is not considered to be very essential and it is not useful for testing.

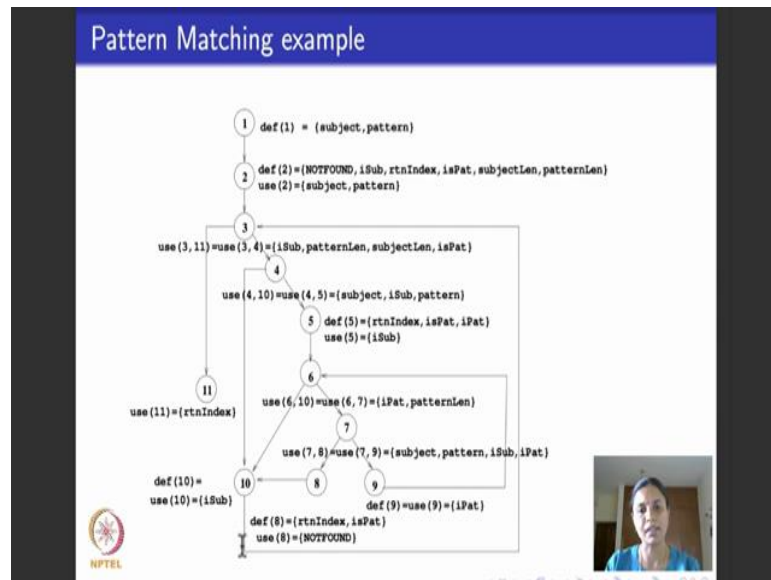
The other kind of grouping that we will do as per definition and use. So, such a grouping is what is called a def pair set. So, we have as usual fixed a variable v . The definition of that variables begins at n_i and its use happens with reference to this node n_j . So, what is a def pair set? Def pair set is a set of du-paths for a variable v that is fixed which begin at node n_i and end at node n_j right. So, a def pair set collects together all ways to get from a definition of a variable which is at node n_i to its use which is at node n_j .

Please remember when we talk about du-paths we had insisted in the last lecture that such path be simple. So, d u paths are always simple. So, def pair set is a collection of such simple paths. A def pair set could also be defined to begin at node n_i , in which case it is the union of all the def path sets for that definition. That is if you say a def path set begins at node n_i for a variable v , then we consider all the different uses at which it can end and take the union and then we say that is a def pair set.

So, just to recap what are we going to do ? We are going to define data flow coverage criteria, by grouping together d u paths. We group together du-paths in 2 different ways the first grouping is as per the definition. We say for a variable v , we collect all du-paths that begin at node n_i , that is the def from variable v is that node n_i . So, that is called a def path set.

The next grouping is as per definition and use. So, for a given fixed variable v , we consider all the beginning at a variable n_i , and all the ending or use at node n_j . That is called a def pair set is a collection of pairs (n_i, n_j) , such that v is defined at n_i and used at n_j .

(Refer Slide Time: 06:15)





So, you remember last time we had looked at the small pattern matching example. It searches for a pattern occurrence in the subject. Actually I would like to mention here that there was a small error in the control flow graph that I had shown you last time. This edge from vertex 10 was drawn to meet at vertex 4 in the control flow graph that I had shown you last time. That is slightly different from the way it occurs in the code. It should actually be like this. This edge at vertex 10 should actually go and meet at vertex 3, this is where the while loop begins.

So, this is the corrected CFG. Please consider the CFG to understand the example. So, if you remember when I had first drawn the CFG, I had given you labels with all the statements and I taken the same CFG and annotated the graphs with definitions and uses. So, that is what this graph is I have put the same graph with this small correction done. So, what I will do is we will understand how the various du-paths look like.

(Refer Slide Time: 07:19)

Def-paths and def-pairs: Example

- In the pattern matching example, there is a definition of iSub at node 10.
- The following is the du-path set with respect to iSub at node 10: $du(10, iSub) = \{[10,3,4], [10,3,4,5], [10,3,4,5,6,7,8], [10,3,4,5,6,7,9], [10,3,4,5,6,10], [10,3,4,5,6,7,8,10], [10,3,4,10], [10,3,11]\}$
- The above def-path set can be split into the following def-pair sets:
 - $du(10,4,iSub) = \{[10,3,4]\}$.
 - $du(10,5,iSub) = \{[10,3,4,5]\}$.
 - $du(10,8,iSub) = \{[10,3,4,5,6,7,8]\}$.
 - $du(10,9,iSub) = \{[10,3,4,5,6,7,9]\}$.
 - $du(10,10,iSub) = \{[10,3,4,5,6,10], [10,3,4,5,6,7,8,10], [10,3,4,10]\}$.
 - $du(10,11,iSub) = \{[10,3,11]\}$.



For simplicity I have taken this variable isub that is the variable that I have fixed there is a definition of this variable isub at node 10, if you see here isub definition at 10 variable isub is defined.

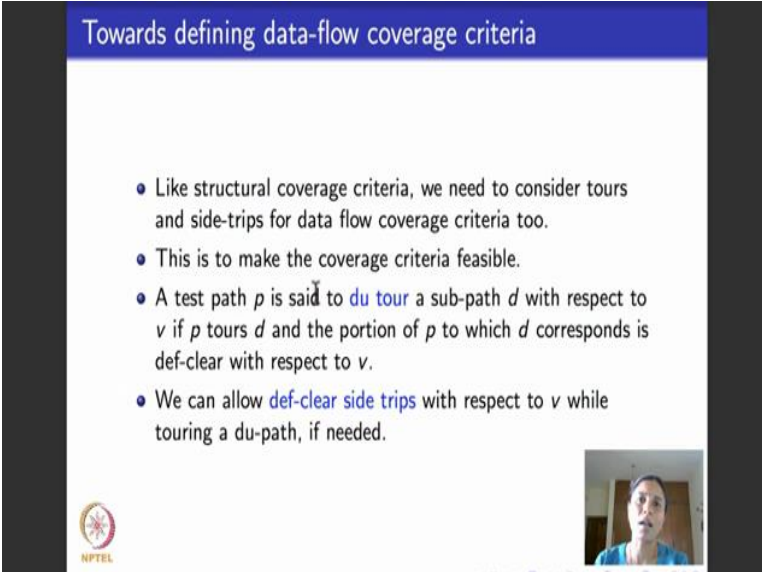
What is isub? If you remember in the code it is an index that runs over the subject right and tries to pattern match every character in the subject with the corresponding character in the pattern. So, here is a definition of isub. So, I can group this definition of isub in two different ways at node 10. I can group it with reference to all it is uses at node 10 or I can group it with reference to it is definition at node 10, and use at each of the other nodes where it is used. If you see it is used in this particular node, it is used in this particular edge, it is used in this particular node, it is also used in this particular edge from 7, 8 to 7, 9. So, there are several places.

So, if I try to trace the def path sets and the du-path sets. So, here are the du-path sets for isub at node ten. So, from 10 I can go back through this edge to 3 and then to 4 right that is what this trace is here and then from 10 to 3 to 4 to 5. So, 10 to 3 to 4 to 5 and then I can do 4 to 5, 5 to 6, 7 to 8 or 7 to 9 both of them have isub in them. That is what these 2 sets paths do 3 and 3, 4, 5, 6, 7, 8, 10, 3, 4, 5, 6, 7, 9 and so on.

So, basically what I am saying here is that you fix this variable isub. You begin its definition at 10, and look at all the places where it is used. It is used in this edge, it is used in this edge, it is used in this edge. So, trace path trace paths, in the graph that begin

it is definition at 10 and end in one of it is uses take all their union that is the set. Alternatively, I can say I begin its definition at 10 and consider its use at 4, within which case I get only this path. When I say I begin its definition at 10 and consider its use only at 5. I get this path beginning at 10 ending at 5. Similarly, from beginning at 10 ending at 8 is this path, beginning at 10 ending at 9 is this path and so on right. So, if I take the union of all these things which begin at 10 and end at various vertices, I get this set right. So, this is how I define def path set and def pair set for a variable at a particular node and in the control flow graph.

(Refer Slide Time: 10:04)



Towards defining data-flow coverage criteria

- Like structural coverage criteria, we need to consider tours and side-trips for data flow coverage criteria too.
- This is to make the coverage criteria feasible.
- A test path p is said to **du tour** a sub-path d with respect to v if p tours d and the portion of p to which d corresponds is def-clear with respect to v .
- We can allow **def-clear side trips** with respect to v while touring a du-path, if needed.

NPTEL

Now, we are ready to define data flow coverage criteria. So, like structural coverage criteria if you remember while we had defined structuring coverage criteria I had told you that sometimes structural coverage criteria can get to be infeasible because sometimes certain kinds of loops might insist that a path be traversed at least once. So, we do test path to trace coverage criteria by taking side trips and detours. That is what we will do here. We say a test path p is said to du-tour sub path d as long as it tours the sub path d , and this sub path that it toours is definition free for that variable.

Remember the only thing that we insist is the defines once it defined at a particular node, it reaches it is use and at an other node it could be through a direct test path or it could be through a test path with the side trip or a detour, irrespective of whether I take the side trip or a detour I insist that that side trip or detour also be definition clear, that is what

this says and it says you can freely use side trips and detours, whenever you want to get test paths to satisfy coverage criteria this is exactly like we did for structural coverage criteria.

(Refer Slide Time: 11:17)

Data flow criteria

There are three common data flow criteria:

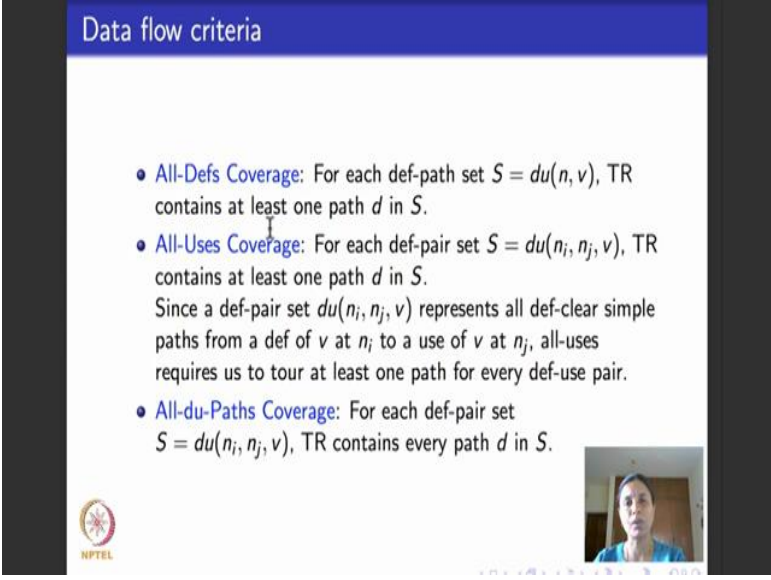
- TR: Each def reaches *at least one use*.
- TR: Each def reaches *all possible uses*.
- TR: Each def reaches all possible uses through *all possible du-paths*.

To get test paths to satisfy these criteria, we can assume best effort touring, i.e., side trips are allowed as long as they are def-clear.

Now, what are the 3 kinds of data flow coverage criteria that we are going to look at. These are the three TRs for the three different data flow coverage criteria that we are going to look at. The first coverage criteria says that each definition should reach at least one use. The second coverage criteria says that each definition should reach all possible uses. The third coverage criteria says that each definition reaches all possible uses not only does it reach all possible uses; it reaches all possible uses using all possible different paths that you can trace out in the graph.

So, as I told you these suggest the test requirements. Whenever you need to get test paths to satisfy these test requirements, you can assume what is called best effort touring. Best effort touring basically says that feel free to allow side trips and detours if you want to make these test requirements feasible.

(Refer Slide Time: 12:15)



Data flow criteria

- **All-Defs Coverage:** For each def-path set $S = du(n, v)$, TR contains at least one path d in S .
- **All-Uses Coverage:** For each def-pair set $S = du(n_i, n_j, v)$, TR contains at least one path d in S .
Since a def-pair set $du(n_i, n_j, v)$ represents all def-clear simple paths from a def of v at n_i to a use of v at n_j , all-uses requires us to tour at least one path for every def-use pair.
- **All-du-Paths Coverage:** For each def-pair set $S = du(n_i, n_j, v)$, TR contains every path d in S .

NPTEL

So, here are the definitions of the coverage criteria. What is all defs coverage say? It says that for each definition of a variable v at a node n the test requirement contains at least one path that reaches a use. All uses coverage says that for a variable v that is defined at a node n_i , I reach one path which basically reaches all the uses for every pair of def at n_i and use at n_j .

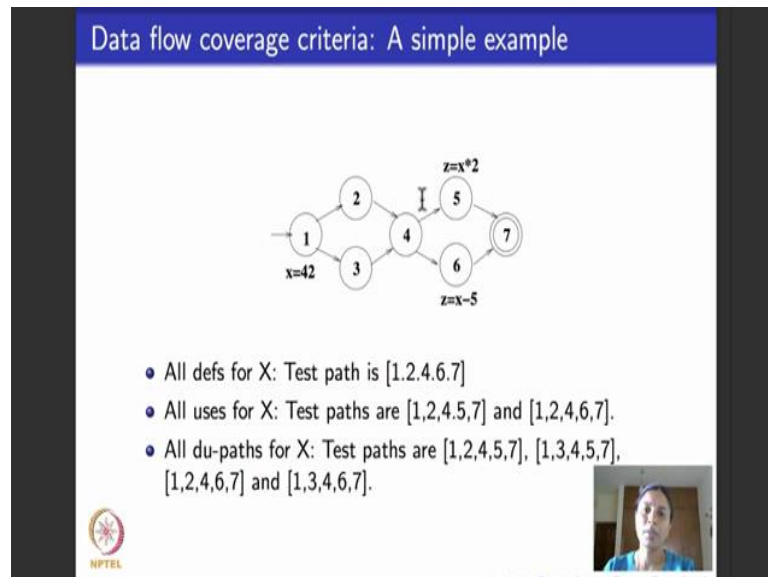
So, to repeat, what is all uses coverage? Fix a variable v , variable v is defined at n_i and used at n_j . And this is, for every possible use of the variable v . So, n_j where is over different uses of the variable v and my test requirement says you cover every possible path, that we takes the variable v , v from it is definition at n_i to it is use at n_j .

The third one says all du path coverage, it says for each def pair set $n_i n_j v t r$ contains every path d in s . If you find these definitions cumbersome the easiest way to understand data flow criteria is to look at the previous slide. So, it says there is one criteria with says every def reaches at least one use, that is all defs coverage. I do not worry about covering all the uses, but I want to cover every definition.

The second one, all uses coverage says every definition reaches all possible uses. So, that is why it is called all uses coverage. The third definition all du -paths coverage says therefore, every definition and every possible use of it take different paths every possible path that goes from a definition to use. Right, is it clear? So, basically 3 different elementary data flow criteria cover every definition make sure it reaches at least one use.

The second says cover every definition make sure it reaches all it is uses. The third says cover every definition make sure it reaches all it is uses every possible different paths to get to the uses.

(Refer Slide Time: 14:31)



So, you remember this small example that we have seen the last lecture. There were only 2 variables in this small graph x and z. x was defined here and used a 5 and 6, z was defined at nodes 5 and 6. So, suppose I have to cover all defs criteria for x. Then the test path that I would take is this 1, 2, 4, I could either take 6, 7 or 5, 7. Basically what I want to say is that x is defined at node 1, take a test path that covers this definition and one use. It could cover either this use at 5 or it could cover this use at 6. The example test path that we have given covers the definition at 1 and the use at 6.

The second criteria, all uses for x basically says x is defined here, and used in 2 different places in the graph at node 5 and at node 6. So, the 2 different test paths that I should take are 1, 2, 4, 5, 7 which covers the use at 5 and 1, 2, 4, 6, 7 which covers the use at 6. Of course, I could take these 2 test paths by going through node 3 that is also the same that will equally well meet the test requirement. In this particular case I have just taken the 2 test path that go through node 2.

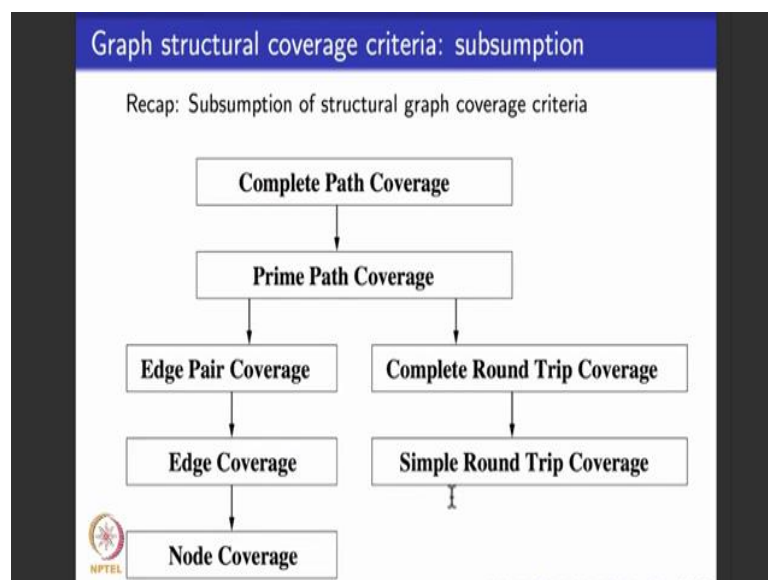
The third condition says from the definition of x at node 1 to which uses a nodes 5 and 6, you not only cover this definition to all it is uses you consider all the paths that take you from this definition to this use. So, if you see there are 4 different possible paths. I can do

this 1, 2, 4, 5, 7 or I can do 1, 3, 4, 5, 7, that will cover the definition of x at node 1 to its use at node 5 right 2 different ways.

Similarly, if I consider the definition of x at node 1, and this used as node 6, there are 2 different paths again.,1, 2, 4, 6, 7 and 1, 3, 4, 6. That is what I have listed here right. Is it clear? So, just to repeat all def says go from every definition to at least one use, or uses says go from every definition to every use, all du-path says go from every definition to every use taking every possible different paths to do this. These are the three elementary graph coverage criteria that we will see.

Now, I want to spend some time trying to make you understand how these coverage criteria are related to each other.

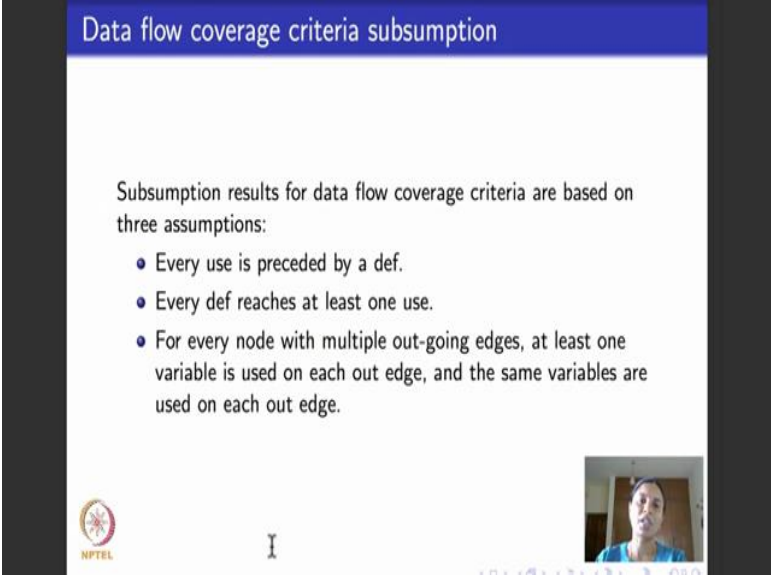
(Refer Slide Time: 17:08)



So, if you remember when we looked at structural coverage criteria we had seen all these coverage criteria and this was the picture that gave how each coverage criterion subsumes the other right. Node coverage, edge coverage subsumes node coverage, edge pair coverage subsumes edge coverage, prime path coverage subsumes all 3 and so on.

Now, we want to be able to add data flow criteria to this picture and understand how the three data flow criteria that we define subsume each other and how are they related to the structural coverage criteria.

(Refer Slide Time: 17:43)



The slide is titled "Data flow coverage criteria subsumption" in a blue header. The main content area is white and contains the text: "Subsumption results for data flow coverage criteria are based on three assumptions:". Below this, there is a bulleted list of three assumptions:

- Every use is preceded by a def.
- Every def reaches at least one use.
- For every node with multiple out-going edges, at least one variable is used on each out edge, and the same variables are used on each out edge.

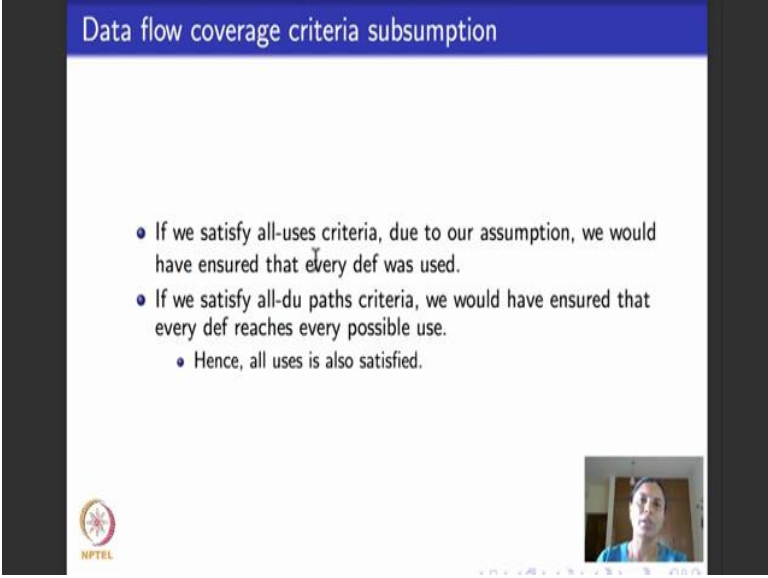
In the bottom right corner of the slide, there is a small video inset showing a person speaking. The NPTEL logo is visible in the bottom left corner of the slide.

So, as far as data flow coverage criteria is concerned before we look at subsumption I would like to make some elementary assumptions. These are not strong assumptions they are basically true for every program that we expect to be compiled and working fine.

So, the assumptions that we are making are every use is preceded by a definition, otherwise you will understand right if there is a use that for a variable that is not defined then compiler will throw an error right especially if the variable is not declared. So, it is not a big restricting assumption this is true of all programs that are syntactically sound and composable. The second assumption that we make is that every definition reaches at least one use. This may not be found by a compiler, but elementary static program analysis tools will be able to find this it, basically says that there are no variables that I define and never used in the program. So, every variable that is defined used at some place in the program.

The third assumption says that when I have a node in the graph which has branches for multiple outgoing edges, at least one variable should be used on each of the out edge and we assume that the same variables are used on each out edge. What it says is that even if different variables are used you consider the set by considering the union of all the variables.

(Refer Slide Time: 18:59)



The slide is titled "Data flow coverage criteria subsumption" in a blue header. It contains three bullet points:

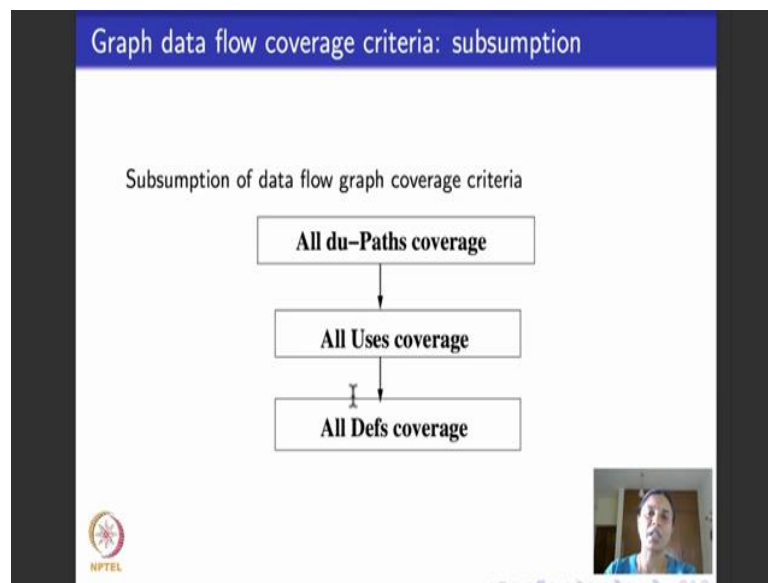
- If we satisfy all-uses criteria, due to our assumption, we would have ensured that every def was used.
- If we satisfy all-du paths criteria, we would have ensured that every def reaches every possible use.
 - Hence, all uses is also satisfied.

The NPTEL logo is in the bottom left corner, and a small video inset of a speaker is in the bottom right corner.

So, under these assumptions we are ready to look at data flow coverage criteria subsumption. What are the subsumption thing? So, there are 3 criteria if you remember what are the 3 criteria all defs coverage, all uses coverage all du-paths coverage. Because all uses says from every def you go to every possible use it is a reasonably straightforward to see that all uses coverage subsume or definitions coverage.

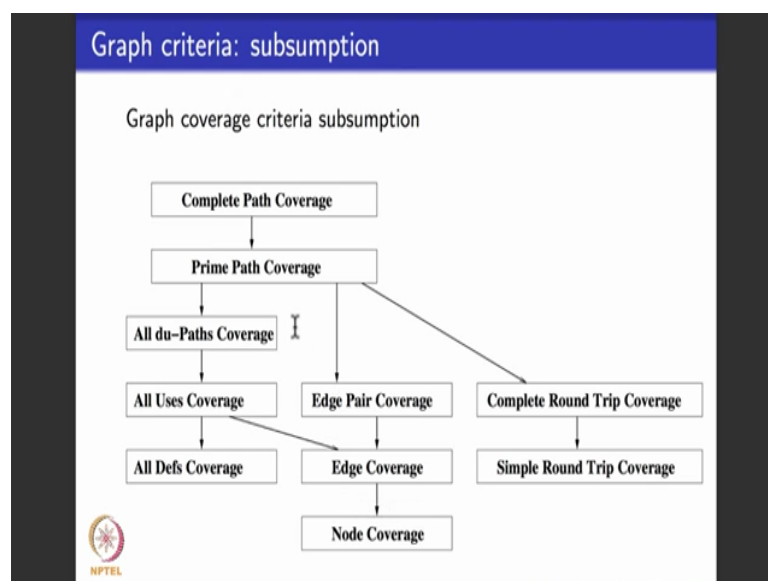
In turn all du-paths coverage says you not only reach every def to every possible use you figure on all possible ways of going from the def to use right. So, it by default as subsumes all uses coverage. So, that is what is given here. It says that if we satisfy all uses criteria by our definition we would have ensured that every definition was used. So, it subsumes all defs criteria again if we satisfy all du-paths criteria, we would have ensure that every definition reaches every possible use. So, it subsumes all uses.

(Refer Slide Time: 20:00)



So, the picture for data flow coverage criteria subsumption looks somewhat like this. All du-path coverage subsumes all uses coverage which in turn, subsumes all defs coverage. So, this is exclusively for data flow coverage criteria. This diagram is exclusively for structural coverage criteria, but both deal with graphs. So, I would be able to want to reach a stage such that I can relate one to the other. That is what we are going to do. Our goal is to understand such a picture.

(Refer Slide Time: 20:31)



So, if you see this is two of the subsumption criteria figures merged into one. On this left hand side that I am tracing out here up through my mouse this path, which begins at complete path coverage prime path coverage goes on to this edge pair edge node and these round trip coverage criteria was purely related to structural coverage criteria. These 3 where the data flow coverage criteria subsumption that we saw now.

What have I done? Now I have put this extra arrow here and then this extra arrow here. So, I have come up with 2 extra subsumption relation. First one says the prime path subsume all d u paths, the second one says all uses subsume edge coverage. These 2 are the only two new subsumption relations that I have put the rest were all explained earlier. Why do these two additional new ones hold? They hold because of the following reason.

(Refer Slide Time: 21:33)

Graph criteria: subsumption

- Each edge of the graph is based on the satisfaction of some predicate. So, each edge has at least one use.
 - Hence, all uses will guarantee that each edge is executed at least once.
 - So, all-uses subsumes edge coverage.
- Each du-path is a simple path, so prime path coverage subsumes all-du-paths coverage.

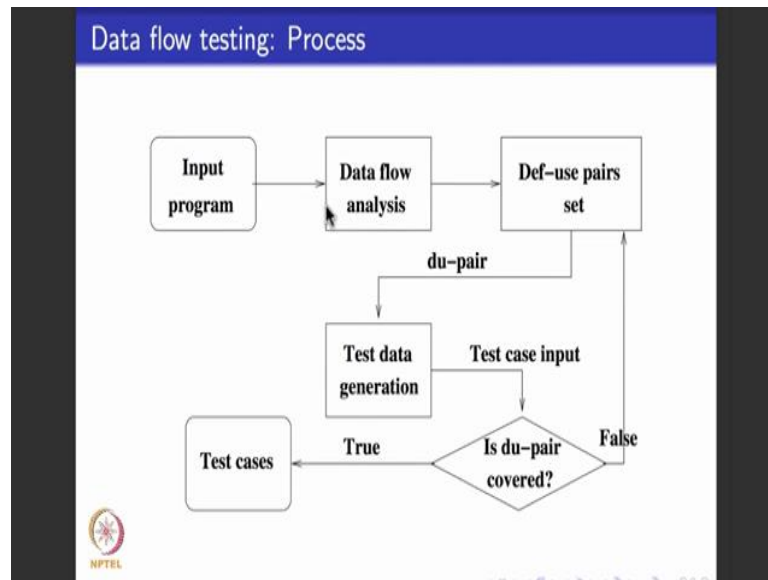
NPTEL

You remember each edge in the graph is based on the satisfaction of some predicate that was one of the assumptions that we meet.

So, each edge has at least one use and because of this when I cover all uses I definitely cover all edges. So, all uses subsumes edge coverage, this arrow. Now because every d u path is a simply path prime path coverage subsumes all du-path because prime paths and simple paths that are not sub paths of any other paths. So, prime path coverage subsumes all du-paths. This is of course, a small point to be noted here is that this this subsumption is with reference to only feasible test criteria, but for now we can safely assume that this holds for most of the programs that we will look at.

Now that we have seen data flow coverage criteria and how the subsumption works and this is the overall picture for graph coverage criteria. I would like to spend some time looking or discussing algorithms for data flow coverage criteria.

(Refer Slide Time: 22:35)

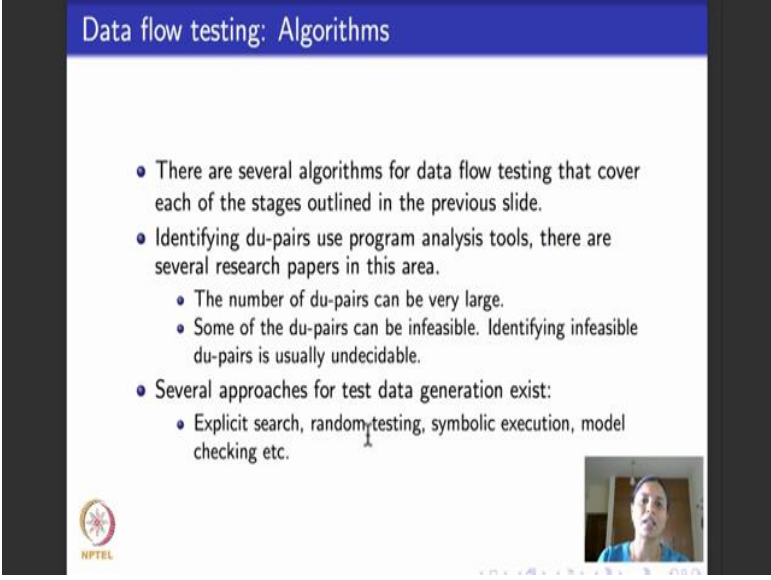


So, when I do data flow coverage this is the overall flow chart of the process for data flow coverage. So, I begin with my input program, I do some kind of a data flow analysis on that program. For this you could use any readymade static program analysis tool available or you could write your own elementary data flow analysis tool. And then using some kind of data flow analysis I pull out the def use pair sets.

Once I have a du-pair my goal is to be able to define some coverage criteria and generate test data for that coverage criteria. If I finish achieving my coverage criteria I am ready with my test cases, otherwise maybe the coverage criteria that I defined was infeasible. I go back, pull out another du-pair and attempt all over again to work with a new coverage criteria that would again be feasible or infeasible.

So, this part, data flow analysis is not in the scope of this course. So, I will leave you to read this on your own. Feel free to pick up any basic books related to the first few chapters of compilers or use some elementary static program analysis tools to be able to do this. We will try to do a brief discussion on what are the algorithms for this part.

(Refer Slide Time: 23:51)



Data flow testing: Algorithms

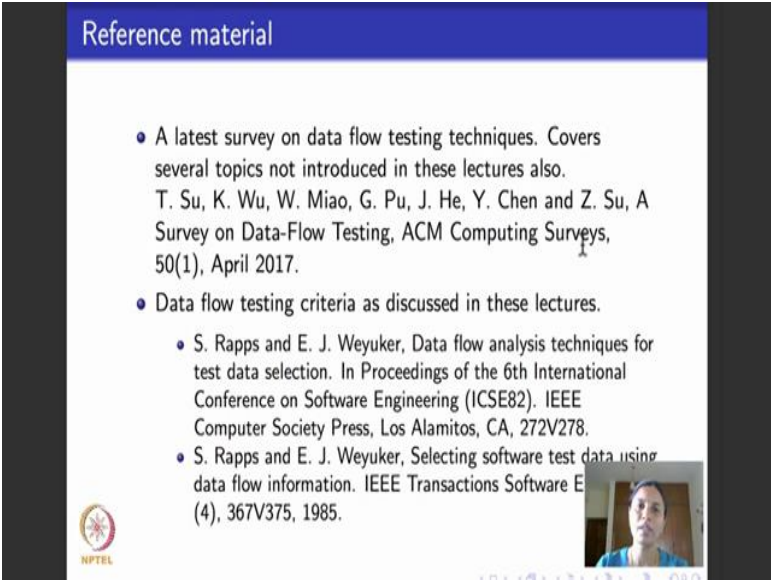
- There are several algorithms for data flow testing that cover each of the stages outlined in the previous slide.
- Identifying du-pairs use program analysis tools, there are several research papers in this area.
 - The number of du-pairs can be very large.
 - Some of the du-pairs can be infeasible. Identifying infeasible du-pairs is usually undecidable.
- Several approaches for test data generation exist:
 - Explicit search, random testing, symbolic execution, model checking etc.

NPTEL

Video inset showing a person speaking.

It so happens that for test data generation there are. So, many different algorithms you can do algorithms based on explicit search, based on random search, based on symbolic execution, based on model checking several different techniques and about four decades of research has gone into algorithms in these areas. I will not spend time looking at these various algorithms, because we would like to move on looking at other testing test case definition terminologies.

(Refer Slide Time: 24:18)



Reference material

- A latest survey on data flow testing techniques. Covers several topics not introduced in these lectures also.
T. Su, K. Wu, W. Miao, G. Pu, J. He, Y. Chen and Z. Su, A Survey on Data-Flow Testing, ACM Computing Surveys, 50(1), April 2017.
- Data flow testing criteria as discussed in these lectures.
 - S. Rapps and E. J. Weyuker, Data flow analysis techniques for test data selection. In Proceedings of the 6th International Conference on Software Engineering (ICSE82). IEEE Computer Society Press, Los Alamitos, CA, 272V278.
 - S. Rapps and E. J. Weyuker, Selecting software test data using data flow information. IEEE Transactions Software E (4), 367V375, 1985.

NPTEL

Video inset showing a person speaking.

But what I would like you to point out in this good survey that is come out with the recently dated April 2017, available as one of the ACM computing surveys. That is an exhaustive survey on data flow technique testing techniques and you will be able to find reference to papers that use all these approaches for test data generation in this survey.

Most of the data flow testing criteria that we discussed in this paper has been borrowed by these these papers. So, Weyuker and her student Rapps. So, these two are very classical papers for data flow testing techniques. As you see they are dated early 80s and quite exhaustively refered to. The material that I have presented is basically derived from these 2 papers and the textbook on software testing by Ammann and Offutt. The next module we will model graph source code as graphs and we will see how the various structure coverage criteria, that we saw till now can be used to test code and then we look at design requirements and by next week I hope to finish the module on graph based testing.

Thank you.

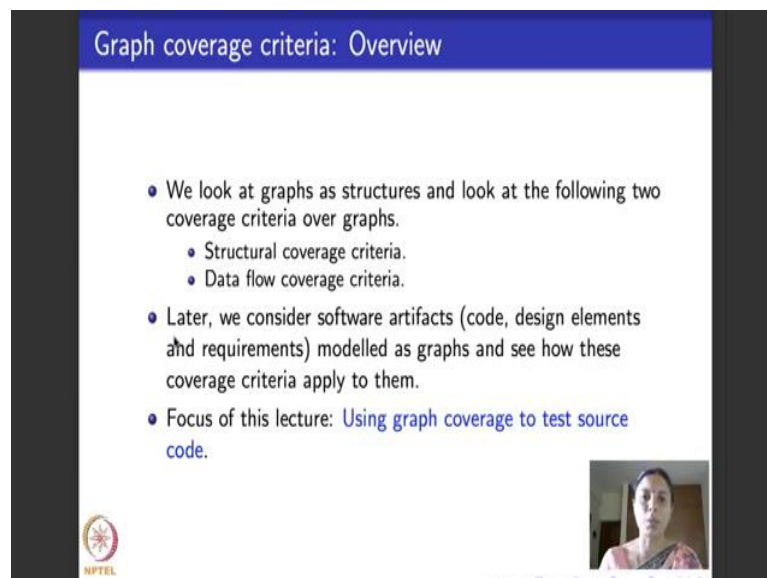
Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 13
Graph coverage criteria: Applied to test code

Hello again, we are in week three. What I will be doing today? We will actually begin to do real testing algorithms today. So, what we saw till now? We saw various coverage criteria over graphs structural or control flow coverage criteria, and then we saw data flow coverage criteria. While doing that we did two things--- we considered graphs as just models and defined the coverage criteria, saw what they mean and also discussed their subsumption relations. And while doing that I gave you a few examples, but I did not really tell you how to use these graph based coverage criteria to actually test software artifacts.

So, what we will do from today's lecture onwards is to consider software artifacts one after the other and then see how they can be modeled as graphs, and how we can use the various graph coverage criteria that we have learnt to be able to actually test the software artifacts. The first software artifact that I will begin with is code - source code because that is the most commonly available source software artifact, in fact it is the most exhaustive software artifact apart from testing.

(Refer Slide Time: 01:12)



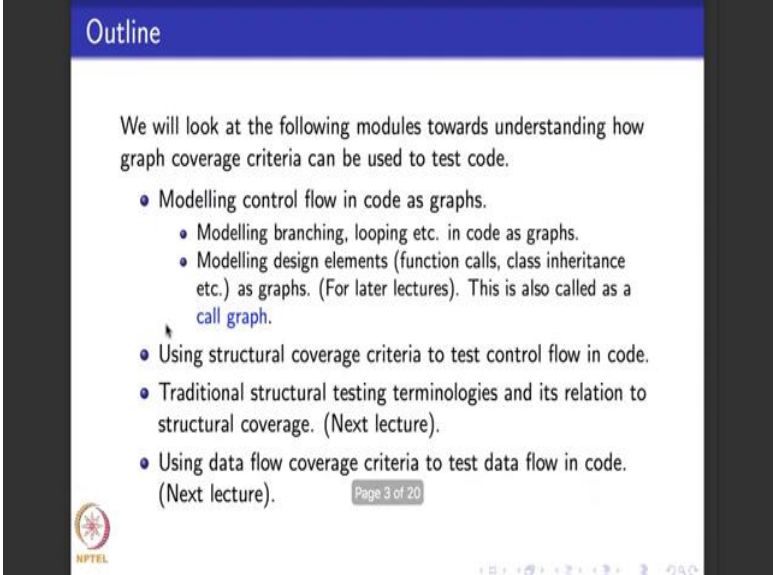
Graph coverage criteria: Overview

- We look at graphs as structures and look at the following two coverage criteria over graphs.
 - Structural coverage criteria.
 - Data flow coverage criteria.
- Later, we consider software artifacts (code, design elements and requirements) modelled as graphs and see how these coverage criteria apply to them.
- Focus of this lecture: Using graph coverage to test source code.

The slide also features the NPTEL logo in the bottom left corner and a small video inset of Prof. Meenakshi D'Souza in the bottom right corner.

So, we will take source code we will see how to model it as graphs, and then we will look at one after the other, the various coverage criteria that we have learnt and see how we are going to apply them to actually test the source code.

(Refer Slide Time: 01:41)



The slide is titled "Outline" in a blue header. The main content is a list of topics to be covered, starting with an introductory sentence. The topics are listed as bullet points. The first bullet point is "Modelling control flow in code as graphs.", which has two sub-bullets: "Modelling branching, looping etc. in code as graphs." and "Modelling design elements (function calls, class inheritance etc.) as graphs. (For later lectures). This is also called as a call graph." The second bullet point is "Using structural coverage criteria to test control flow in code." The third is "Traditional structural testing terminologies and its relation to structural coverage. (Next lecture)." The fourth is "Using data flow coverage criteria to test data flow in code. (Next lecture)." There is a small NPTEL logo in the bottom left corner and a "Page 3 of 20" indicator in the bottom right corner.

Outline

We will look at the following modules towards understanding how graph coverage criteria can be used to test code.

- Modelling control flow in code as graphs.
 - Modelling branching, looping etc. in code as graphs.
 - Modelling design elements (function calls, class inheritance etc.) as graphs. (For later lectures). This is also called as a call graph.
- Using structural coverage criteria to test control flow in code.
- Traditional structural testing terminologies and its relation to structural coverage. (Next lecture).
- Using data flow coverage criteria to test data flow in code. (Next lecture).

NPTEL

Page 3 of 20

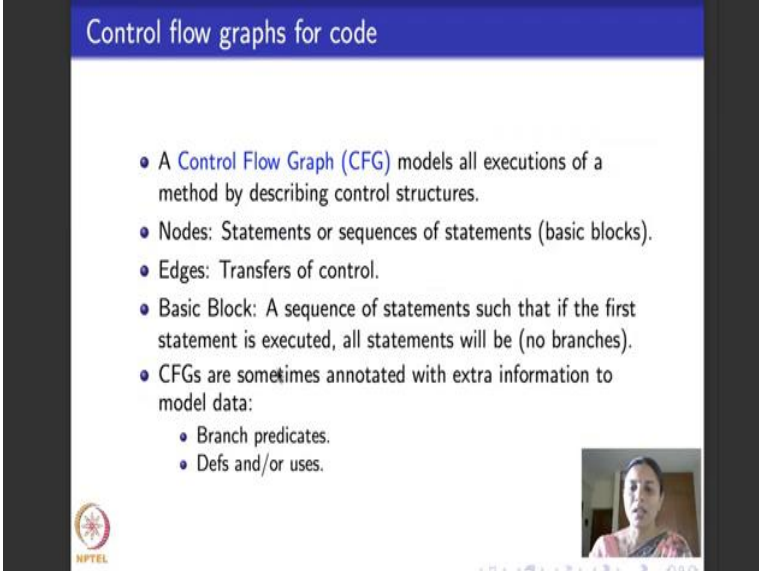
So, when it comes to source code, how are we going to model source code as graphs? The most common model of source code for graphs is the control flow graph right. So, control flow graph typically models branching, looping, you know sequential calls of statements and so on and so forth. It also models things like function calls, class inheritance and so on. So, these are called call graphs or inter procedural call graphs some times and when you club them together along with the control flow graph you could call it as an inter procedural control flow graph.

What we will see today is to take one, assume that the code consists of one method, one procedure or one function not several different function calls, functions calling each other. We just take this we will try to model it, first see how to model it is a control flow graph, and then see how to use that a structural coverage criteria that we have learnt to be able to test various aspects of this code. In subsequent lectures, I will teach you how to augment this control flow graph with information about data, in particular definition and uses, and how to use the data flow coverage criteria to test these graphs.

In between, you might be wondering, I know testing a little bit, and I know several common terminology is in testing. So, what we will do is we will spend one lecture

looking at what are the common terminologies that I used when it comes to testing source code, and source code with graphs and see how what we are doing relates to what is commonly understood.

(Refer Slide Time: 03:22)



The slide is titled "Control flow graphs for code" in a blue header. It contains a bulleted list of definitions for Control Flow Graphs (CFGs). In the bottom right corner, there is a small video inset showing a person speaking. The NPTEL logo is in the bottom left corner.

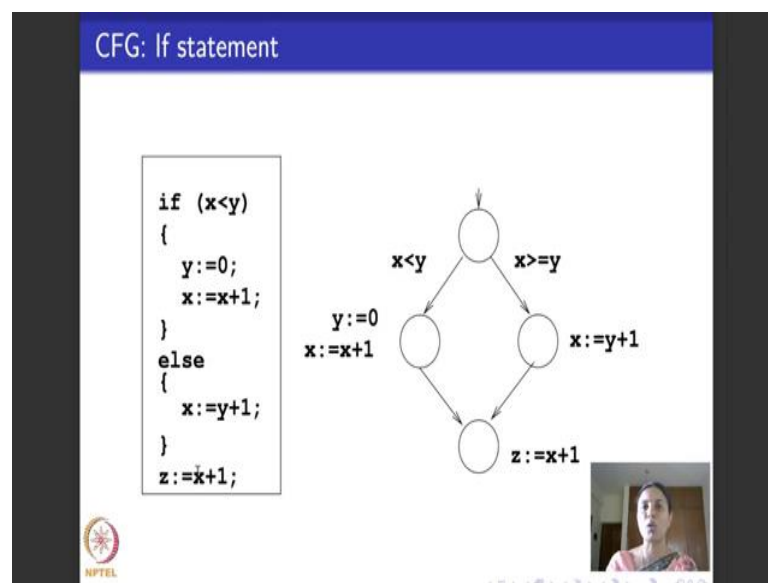
- A **Control Flow Graph (CFG)** models all executions of a method by describing control structures.
- **Nodes:** Statements or sequences of statements (basic blocks).
- **Edges:** Transfers of control.
- **Basic Block:** A sequence of statements such that if the first statement is executed, all statements will be (no branches).
- CFGs are sometimes annotated with extra information to model data:
 - Branch predicates.
 - Defs and/or uses.

So, what will we be doing today? We will be doing following. We will take source code the code that we are going to test; I will assume without loss of generality that the code is contained within one method or one procedure does not have too many procedure calls. We will look at that when we look at design later on. And I will also assume when I do examples that you are familiar with programming languages like C or Java right, the basic how to read code. So, I will tell you how to take this code model this code as a control flow graph and then how to look at structural coverage criteria that we defined over this control flow graph abbreviated as CFG.

So, graph as we know has nodes and edges. So, what are nodes in the control flow graph modeling a code, nodes are basically statements or sequences of statements. They are commonly known as basic blocks, in the sense that is one continuous sequence of statement such that there is no branching in between, there is no if, there is no while, there is no for loops. It just may be series of assignments, a series of statements like assignments and printf's, series of asserts something like that, it just a continuous sequence of statements commonly known as basic blocks.

And then what are edges, edges basically dictate transfer of control. So, we see it through examples in detail. So, control flow graphs are many times annotated with what are called branch predicates which tell you which is the predicate on which you are branching. It comes as labels of the edges, they are also annotated with defs and uses of variable as we saw in the example. For the purposes of today's lecture we will look at examples, we look at annotations where branch predicates come. This defs and uses which deal with augmenting control flow graph with data, I will deal with it separately in another lecture.

(Refer Slide Time: 05:13)



So, what we will do from now on for the next few slides is I will take each kind of software structure and tell you how with control flow graph corresponding to that software structure looks like. So, here is how control flow graph corresponding to a if statement will look like. So, let us say we have this code snippet. I told you when I show you one example like this, I am just showing you a fragment of the code, this is not a complete piece of code, definitely not compilable or executable as a standalone entity. We just look at a fragment because our focus is to understand for the concerned fragment how its control flow graph will look like.

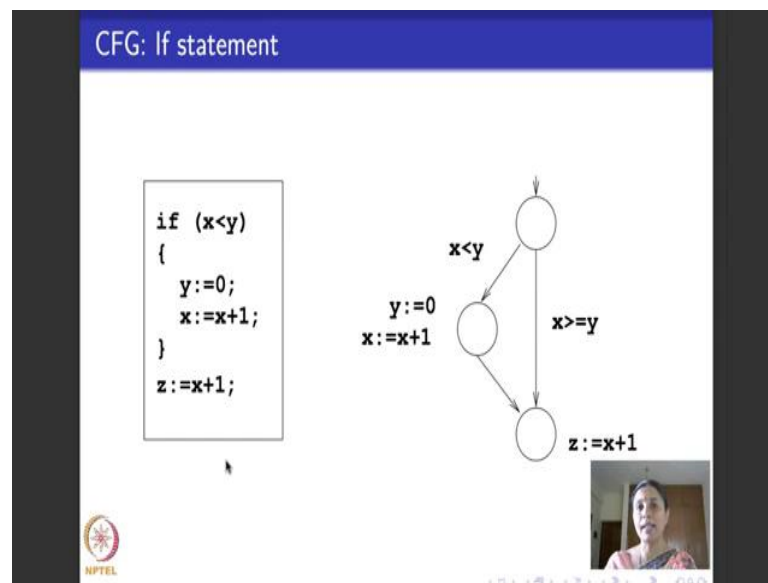
Later we will take the full piece of code, a large example that contains all the statements and we will put together the CFGs that we learnt to draw for each of this fragments and see how the CFG for the whole code will look like right. So, what I will do is I will look

at various code structures like branching, looping, exception handling and so on and tell you how the CFGs for each of those code structures will look like. So, we begin with if statement because that is the simplest. So, let us say we had a code fragment that look like this. So, there is a statement which says if x is less than y , then you do these two statements--- assign 0 to y and make x as x plus 1 else which means if x is greater than or equal to y , then you say x is y plus 1. And irrespective what you do when you come out of this if statement, you say z is x plus 1.

So, how does a CFG for this if statement look like. CFG looks like this. So, this node which is the initial node, what does it say. It corresponds to this if statement. So, you checks for this condition is x less than y . Suppose x is less than y then it take this branch or this edge where it executes these two statements which correspond to these two statements on the code. And suppose not, suppose x is not less than y which means x is greater than or equal to y then it executes this statement, it says x is equal to y plus 1. Irrespective of whichever statement it executes when it comes out of the if loop control flow structure it executes this assignments statement given here in the graph z is x plus 1. So, here is how a control flow graph or a CFG corresponding to an if statement looks like.

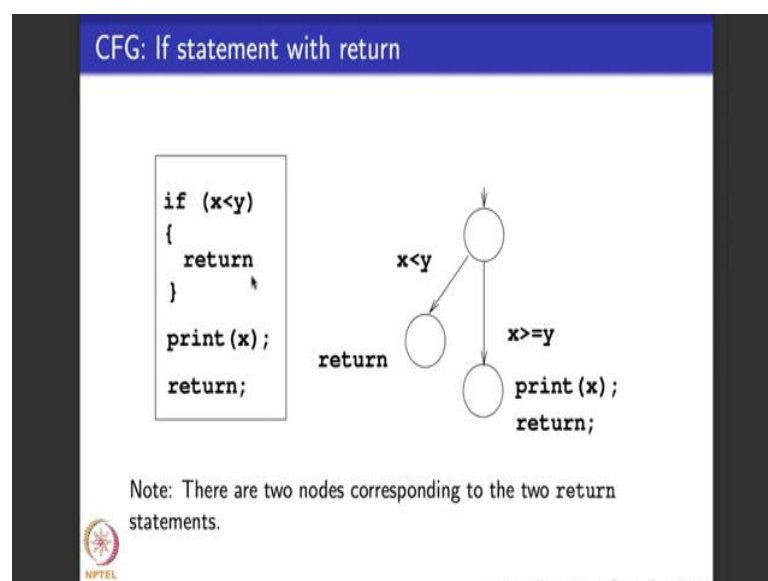
So, there is a node that corresponds to this check or the if statement, if that check is positive let us say if x is less than y , then it takes one branch depicted by an edge and executes this statements corresponding to that branch. If this condition is false then it takes other different branch which represents the negation of that condition; in our case it is a x greater than or equal to y , and it goes to a block where it executes the statements corresponding to the else branch. Whatever it is, it has to come back and merge for this kind of an example and executes the statement that is just outside if statement; in our case it is z is equal to x plus 1. So, I hope this is clear.

(Refer Slide Time: 08:17)



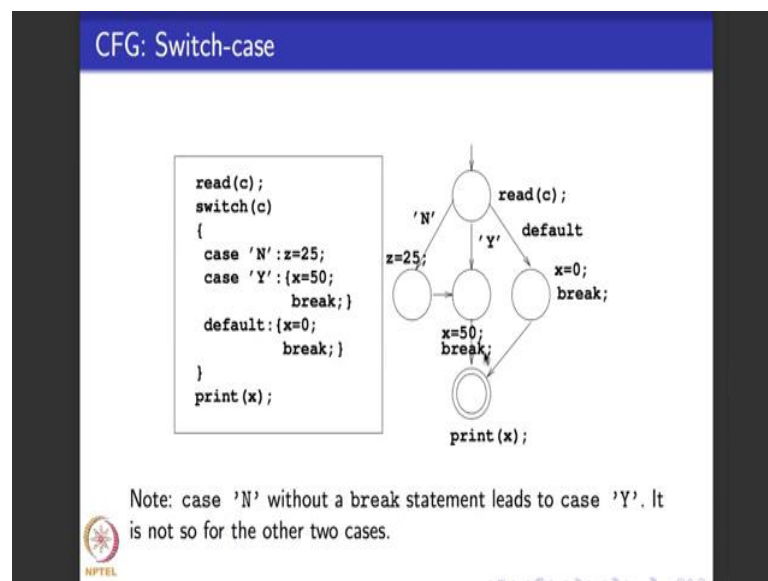
What we will do is suppose we had just if statement without an else part. So, in this example, we had a then part and an else part explicitly written out. Suppose, you just had an if statement which had a then part right I just remove the else from the previous example. How does its CFG look like? So, as you show let begin at node where I check for the truth or falsity of predicate x less than y. If x is less than y then I do this otherwise I just exit directly because there is no else part specified, I directly exit and come out to the node that represents the execution of statement that immediately follows the if statement.

(Refer Slide Time: 08:55)



So, now suppose I had an if statement with, but with this specially marked statement called return. So, let us look at this code fragment here. What it says is that if x is less than y then you return; and when you return you print the value of x because that is the statement that is immediately after this; otherwise you just return. This is some silly piece of code do not worry about what it is meant for and all. Our purpose is to understand how its control flow graph looks like. So, as always I start with one node representing this if statement where I check for truth or falsity of this predicate. Let us say it turns out to be true, I will come here, take this branch and then I do a return statement. Suppose, it is false then I come out, do a print x and then do a return statement. One thing to be noted is that please note that this return is different from this return. So, this is put by distinguishing two nodes corresponding to the two different returns that are there, one inside the if statement and one outside the if statement.

(Refer Slide Time: 09:58)



So, the next kind of branching that we will be looking at is what is switch case statement, it is another very common branching statement. So, let us look at the code fragment. What it says is that you read a value into a variable called c, c looks like a string variable and then you switch to this different cases based on the value of c. If c happens to be n the string n then you execute the statement z is equal to 25; if c happens to be the string y, then you execute these two statements--- x is 50 and break. Break means break out of the switch case statement. If it is none of these cases which means if it is the default case then you execute these two statements. One of the statements again is a break which tells

you to break out of this switch case statement. Whatever you do when you come out, let say you have a print x statement.



How does the CFG for this statement look like. So, I begin with a node which corresponds to this--- read c. And then I merge it with the same node where the switch c is also taken. I can keep another node which is serially connected to the node corresponding to read c there will be no big difference this CFGs are more or less the same. But in this case I chose to keep one node for both the statements. So, and then the next thing represents the three branches corresponding to the three cases of the switch statement. If case is N, I take this branch and do this statement z is 25; if case is Y, I take this branch and do these two statements x is 50 and break. Please read these labels corresponding to this node, they same to overlap with this edge, but they are labels that annotated this particular vertex. And if it x, if c is not N, if c is not Y then I am in the default case in which case I come out and do these two statements. And when I break, I go back here which is the print x statement that is present in this code.

Now, if you look at this CFG carefully there is one extra edge here. What is that extra edge represent? That extra edge says that the case for N if you see here does not have a break statement. If it does not have a break statement then as per the semantics, the typical semantics switch statement, I go and evaluate the next cases. So, what it says is that when it is the case for N you go ahead and lead it to the case for Y and continue from there on. If I do not want this edge then I explicitly put a break along with z is equal to 25 for the case for N. Because such a break is not there the case for N leads to the case for Y as per the semantics switch statement, and that is the reason why this particular edge exists in the control flow graph.

(Refer Slide Time: 12:48)

CFG: Loops

- There could be various kinds of loops: while, for, do-while etc.
- To accurately represent the possible branches out of a loop, the CFG for loops need extra nodes to be added.

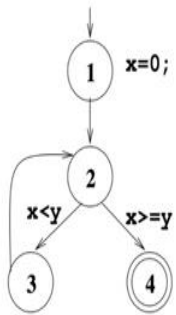


Now, what we will do is we look at loops. When it comes to loops we all know that there are several kinds of loops, there are while loops, there are for loops, there are do-while loops and so on. So, to model loops as CFGs will have to typically add a few extra nodes in the CFG. We will see through examples for each kind of loops how those extra nodes look like, and where are they added, and what do they represent as far as the loops semantics in execution is concerned.

(Refer Slide Time: 13:19)



CFG: While loop

```
x=0;  
while (x<y)  
{  
  y=f(x,y);  
  x=x+1;  
}
```



$y=f(x,y);$
 $x=x+1;$

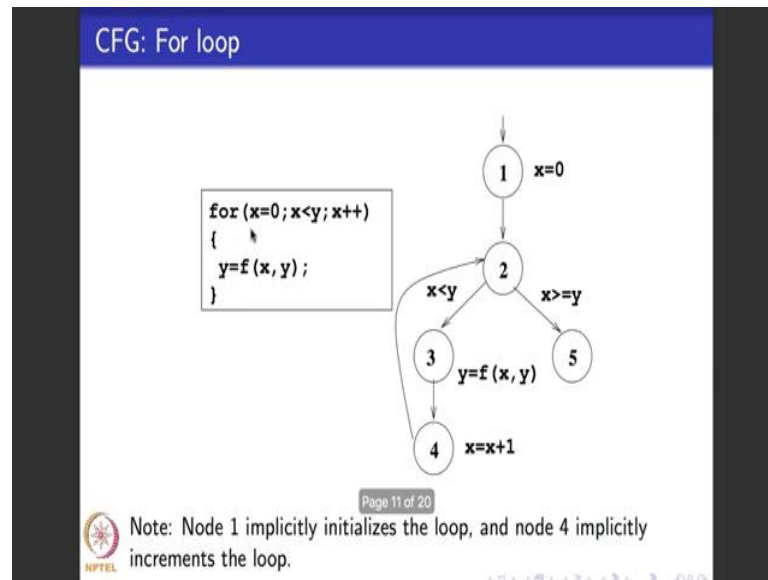
Note: Node 2 in the graph above is a dummy node.



So, we begin with while loop. Let us say you had a simple while loop that looks like this. You initialize x to 0 and then you say as long as x is less than y , you execute these two statements you call a function y . You call a function f with the parameters as x and y and you assign the value that f returns back to y then you do this simple assignment statement which is x is equal to x plus 1. How does the CFG for this code fragment containing a while loop look like? So, initially there is this node which begins for this assignment statement x is equal to 0, then I come and I have to do this node. So, one way of interpreting it is to consider this as a dummy node. Another way of dummy node or an extra node, another way of interpreting it is to consider node 2 as representing this while.

So, while this is true. So, while this predicate is true I go to node 3, where I execute the statements that occur inside the while loop. And I go back to checking for the condition in node 2 which means I go back to checking for the truth or falsity of this predicate. When this predicate becomes false that is when x is greater than equal to y , I exit out of this loop and I stop. And in this particular code, I have not really given you what we are doing when it exits the while loop. So, this does nothing annotating vertex four. And vertex four is marked as a final state because I stop there. So, is it clear please? How do while statement looks like? They will always have this kind of branching structure that is represented by nodes 2, 3 and 4. One branch we will represent the loop condition of predicate being true and it will keep looping as long as the predicate is true that is this branch between two and three. Another branch will represent exiting the loop that is when the loop predicate becomes false.

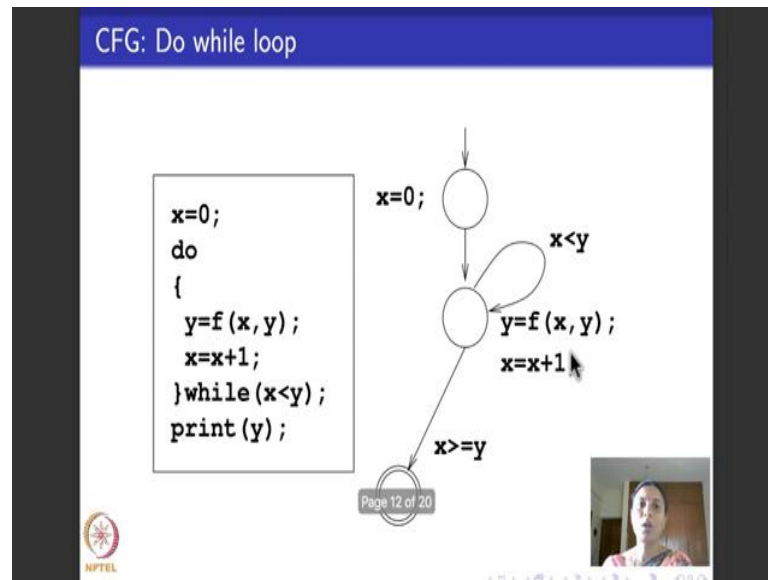
(Refer Slide Time: 15:17)



So, in the next example we look at for loops. So, here is very simple for loop. It says for x is equal to 0 as long as x is less than y , $x++$, you do this condition. So, next how does this look like. So, initially x is equal to 0 as labeled with an initial node. And then here comes this. So, node 1 does not have explicit status in this code fragment if you notice. So, you can think of node one is a dummy node that implicitly initializes the loop; node 2 is the actual check for the predicate x less than y being true. So, if it is true, it goes through node 3 where the statement y is equal to $f(x, y)$ is executed.

Now, after this what happens in the for loop, it is to go ahead and increment x using this $x++$. So, I add another node 4 which is actually a dummy node which implicitly represents $x++$ or x is equal to $x + 1$, it goes back after incrementing x checks whether x is less than y is true; if it is true comes back executes this increments x goes back and so on. So, this is how the for loop goes on between nodes 2, 3 and 4 and the cycle between 2, 3, 4 and 2. So, what happens is the predicate is checked, the statements are executed the variable is incremented predicate is checked again that is keeps repeating and when the predicate becomes false, that is when x is less than or equal to y , the CFG exists the for loop and takes this branch and comes to node 5. So, is it clear please how this we achieve for the for loop looks like.

(Refer Slide Time: 16:56)

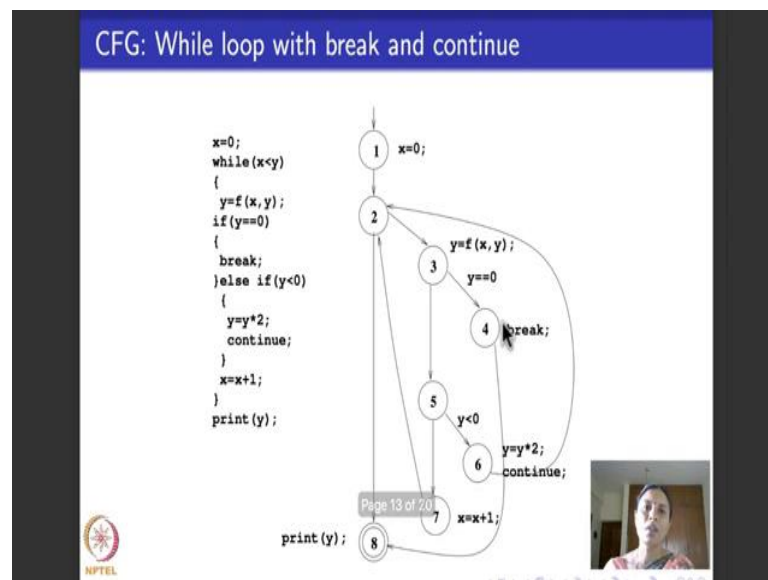


So, now we will move on and look at the control flow graph for a do while loop. So, how does the do while loop work? Unlike a while loop or a while do loop, do while loop will execute the statement inside the loop definitely at least once. It first execute the statements that come inside the loop; after that it checks for truth or falsity of the predicate that labels the loop that is what is depicted in this CFG also. So, here is a code fragment that has a do while loop. I begin with initializing x to 0, then I do the following as long as at the predicate x less than y is true. So, what do I do? I do y is equal to f(x,y) then I do x is equal to x plus 1. And when I come out of this do while loop I do print y right

So, as per the semantics first the statements are executed then this condition is checked, so that is exactly replicated on the CFG. I begin with the x is equal to 0, which corresponds to this statement then I come to a node where these two statements are executed, $y = f(x,y)$, $x = x + 1$ that is these two statements. And then I check for this condition x less than y that is represented as a self loop in this node because as long as this condition predicate x less than y is true. I continue to stay in this node where I execute these statements. And when this condition becomes false, I exit and go to a node where I have to actually do print y; I should have labeled this node as print y right because that is where I go to.

Please note the difference in the control flow structure for a do while loop and for a do, for loop and for a while loop. Say it for a for loop and while loop we had this kind of a branching, where I first check and branch for truth and falsity of a predicate. I go back in a loop whenever the predicate is true. And I branch out whenever the predicate is false same exists for while loop also. I check for the condition in a node I go back to the node as long as predicate is true after executing the statements that label the loop and I branch out and exit the loop when the condition is false. Whereas, for a do while loop, I stay at a particular node as long as the condition is true, and I keep executing this statements of that node as long as the conditions is true and I branch out when the condition is false. So, the loops sort of in this example shrunk to a self loop at one node.

(Refer Slide Time: 19:29)



Now, I will show you slightly bigger example. Here again it is a while loop, but it has two special kinds of statements, statements like break and continue which make loop semantics little more interesting. So, here is a small code segment. So, it says start by initializing x to 0 as long as x is less than y, you execute this large while loop it begins here ends here, the last but one line. What happens inside this while loop? Inside this while loop I do several things I first do this y is equal to f(x, y) by calling the function f and then I check for two nested if statements. So, as a if y is equal to 0, then you break.

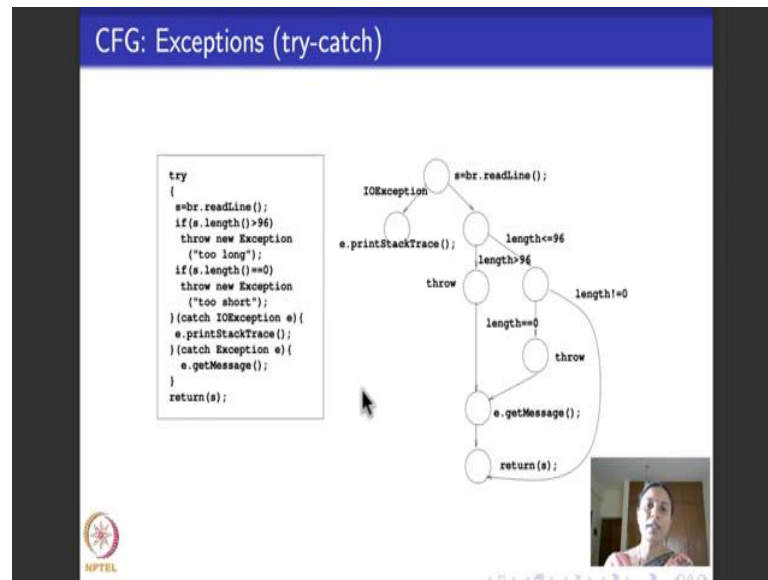
So, what is the semantics of this break? This break means where you will come, you will come here, you come to the print y statement. And then suppose this is not 0 then you

check if y is less than 0, if it is less than 0, and then you do something $y = y+2$ and then you continue. And when you continue implicitly what are you saying when you continue means continue into the y loop, go back and check for this condition that is what it is begin said. So, how does the control flow graph for such a code fragment look like? So, there is an initial node where x is equal to 0 is given, node 2 checks for the predicate x less than y , I haven't labeled the edges here to reduce the clutter in the control flow graph.

But suppose x is less than y is true, then I take this branch to 3, where I execute the statement $y = f(x, y)$; after that I will check if y is equal to 0; if y is equal to 0. Then I break because at break I come out and go to the print f statement, which is that node 8. Suppose, y is non zero then I go here to the else part; in the else part the first thing that I do is to check if y is less than 0; if y is less than 0 then my code says you do these two statements y is equal to y into 2 and then you do continue.

Continue as I told you means that I go back and check the condition of the while loop. Suppose, y was not less than 0, y was greater than 0 then I come out of the second if statement and execute this statement $x = x+1$. But please remember even here I am within the while loop, so I have to go back through this edge from 7 to 2 to the main condition of the while loop node where the condition is checked and I repeat the sequence of executions. So, I hope this makes it clear how this semantics of while combined with if, break and continue works and how the control flow graph corresponding to such statements looks like.

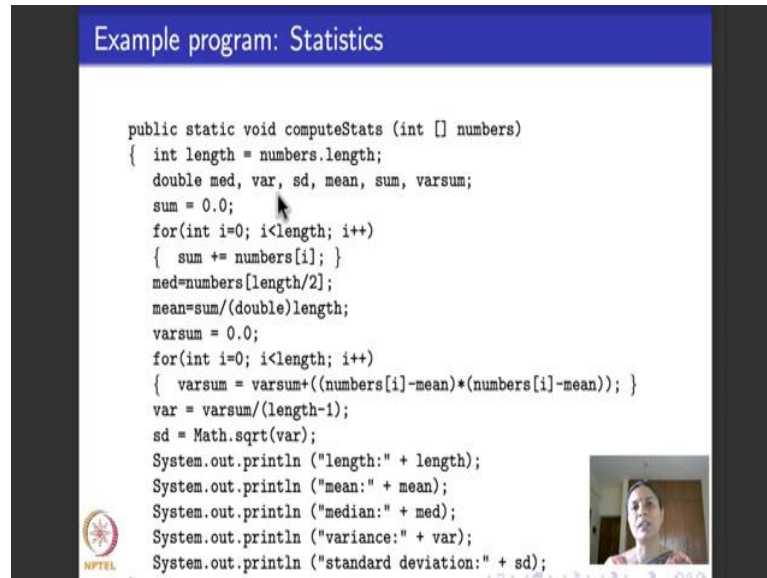
(Refer Slide Time: 22:13)



Now, here is another example of a control flow graph. You might be familiar with this exception handling through using try and catch in Java. So, how does the CFG if such try catch exception handling look like? So here is a again the small code fragment which involves the try catch piece of exception handling. So, it says there is a try here, there are two catches here. So, what it says is that you read the line, assign it to x, and you will see the length of the line that you have the read. If length is greater than 96 then you throw an exception saying is too long.

If length is less than 96 then you check if the length is actually 0; if length is actually 0, then you throw another exception saying too short right. And if it is too short then you go back and get the message; if it is too long then you go and print, if it is not too long sorry then you go and print this stack trace. So, the control flow graph of such a code fragment looks like this. I begin with reading line and assigning into s. Then what I do is I come here and check whether what is the length of the statement line that I have just read. If length is greater than 96, then I throw an exception and go out right to get another message. Suppose length was less than or equal to 96 then I now check if the length 0 or not. If the length is 0 then I throw another exception then I go back to get a message; if a length is non zero then I go and return s. So, this is the how to CFG corresponding to try catch statement looks like.

(Refer Slide Time: 23:47)



Example program: Statistics

```
public static void computeStats (int [] numbers)
{
    int length = numbers.length;
    double med, var, sd, mean, sum, varsum;
    sum = 0.0;
    for(int i=0; i<length; i++)
    {
        sum += numbers[i];
    }
    med=numbers[length/2];
    mean=sum/(double)length;
    varsum = 0.0;
    for(int i=0; i<length; i++)
    {
        varsum = varsum+((numbers[i]-mean)*(numbers[i]-mean));
    }
    var = varsum/(length-1);
    sd = Math.sqrt(var);
    System.out.println ("length:" + length);
    System.out.println ("mean:" + mean);
    System.out.println ("median:" + med);
    System.out.println ("variance:" + var);
    System.out.println ("standard deviation:" + sd);
}
```

So, now, what we have done through all this slides where we have looked till now, is that I have shown you how to through examples how to draw the control flow graph corresponding to several different code constructs. Control flow graphs corresponding to branching which involve if with then and else, without else, if with return statements, control flow graphs corresponding to switch case statement, control flow graph corresponding into different kinds of loops - while loop, for loop, do while loop and CFGs that include while loop with branching and breaks and continue-s. And finally, in this slide we saw an example of a control flow graph for exception handling through a statement like try catch.

What now we move on we will see is I will show you of full piece of code, piece of code that does something, we will draw its full control flow graph and we will see how to use the various data structural coverage criteria that we have learnt to be able to test that source code. So, the code that I want to look at is an example of a program that computes the typical basic entities that deal with statistics like median, variance, standard deviation, mean and so on.

So, what it takes is that it takes an array of numbers and it returns what are the various parameters. So, it returns the length the array, it returns the mean value of the array, it returns the median, it returns the variance and it also returns the standard deviation. So, those are all these println statements. And what is the code do the code basically has to

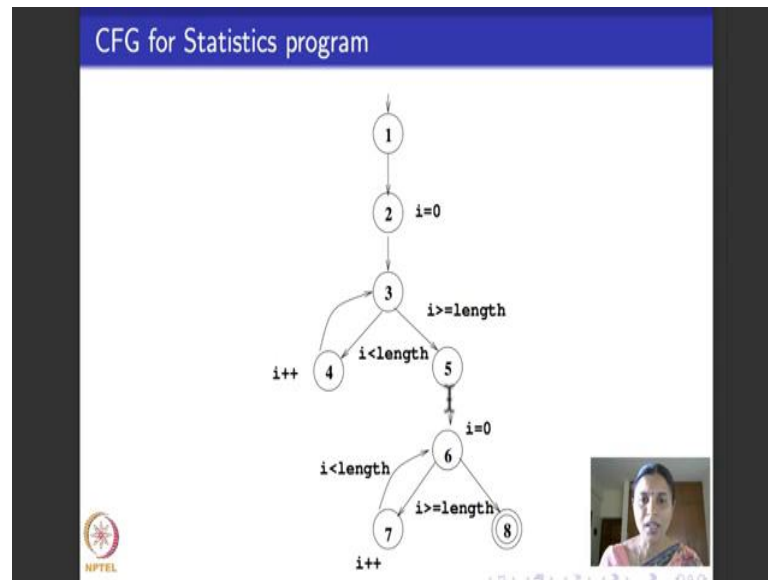
for loops, one for loop right here and another for loop right here. The first for loop is use to compute median and mean, the second for loop is use to compute the variance and the standard deviation. So, I am sorry I put the code in smaller font to make it squeezed into one slide this curling closing brackets is sort of gone down, but this represents the full piece of code. It takes an array of numbers the code is called computeStats. And it outputs length, mean, median, variance and standard deviation.

How does it go about doing it, it says the length is an integer variable and the rest of the numbers are all declared is double, it initializes the sum to be 0. Then it goes into a for loop where it repeatedly adds the number in the array to itself, and computes the sum of all the numbers in this array, in this for loop. Then it says median is this, right, numbers of length by 2 which is the midpoint and it says mean is $\text{sum}/\text{double}(\text{length})$.

So, now this for loop computes the variance sum. So, initializes the varsum to 0 and then it gets into this. And what it does is the there it computes the varsum as this difference which is standard way of doing at in statistics. When it comes out it computes the variance is varsum divided by length of the array minus 1 and standard deviation is the square root of this variance. And then it has all these print statements that print the various values. Now, what is our goal, a goal is to take this piece of code and model it as a control flow graph and use the various structural graph coverage criteria that we have learnt to be able to actually test this code.

So, how am I going to model this code as a control flow graph? So, initially when it comes to CFG, please remember we ignore these entities, we ignore these statements. So, I begin with a node that initializes the sum that is what is given here as node 1. Then the next thing that is there is this for loop; it initializes i to 0 checks for this truth of this predicate and there is a node that increments i, whenever this predicate is true it assigns sum to be this value.

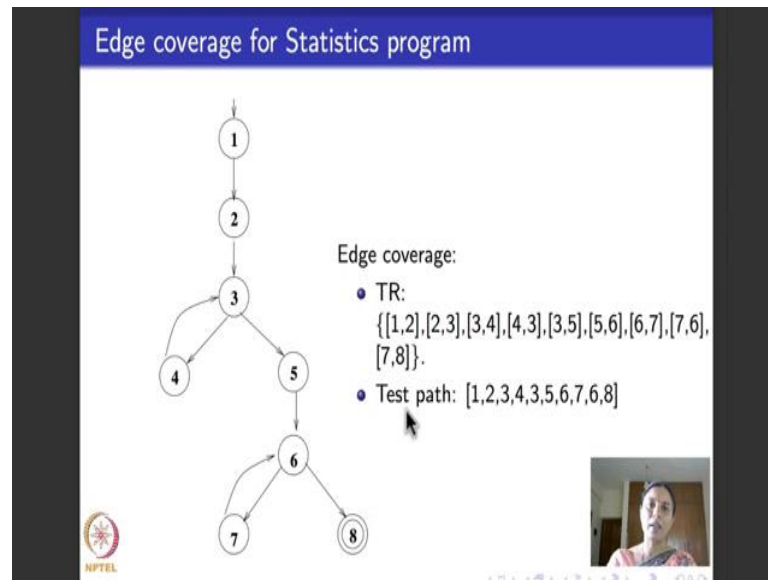
(Refer Slide Time: 27:56)



So, that is represented in this fragment of the CFG. So, it initializes--- nodes 2, 3 4 and 5 represent the first for loop. So, it initializes i to zero node 3 is where it checks whether i is still less than length or i is greater than or equal to length. If i is still less than length it increments $i++$ and adds the thing to sum. If i is greater than length it goes out here and then enters the next for loop which is this loop. So, here again I check for the same this thing and I do this. Just for simplicity and to reduce clutter I have not labeled this control flow graph fully with all these statements. Like for example, I have not used these two statements `median its numbers.length by 2` and `mean is sum of double(length), varsum is 0`. Where should they all come they should all come here with labels of this node, but I wanted to focus only on structural graph coverage criteria.

So, I have just reduced myself to looking at the main CFG corresponding to this one. I have removed all the labels that we are use to annotating the CFG with. I have retained sum just to tell you that this little fragment here is for the first for loop, this is for the second for loop. Ideally speaking I should have put all other labels, but then the CFG would have become to cluttered and the focus on understanding this example for structural coverage criteria will be lost. So, I have just retained as minimal labels as possible, this CFG by no means is complete when it comes to a label annotations. So, now, if taken this code modeled it as a graph.

(Refer Slide Time: 29:57)



Our focus now purely on this graph, right? We want to now look at this graph go back recap all the structural coverage criteria that we have learnt till now and see how we can use some of them to test this graph. So, the first coverage criteria that I would like to, you can apply any of the coverage criteria that we have learnt till now. We will apply a small sampled two or three different coverage criteria for the purposes of understanding.

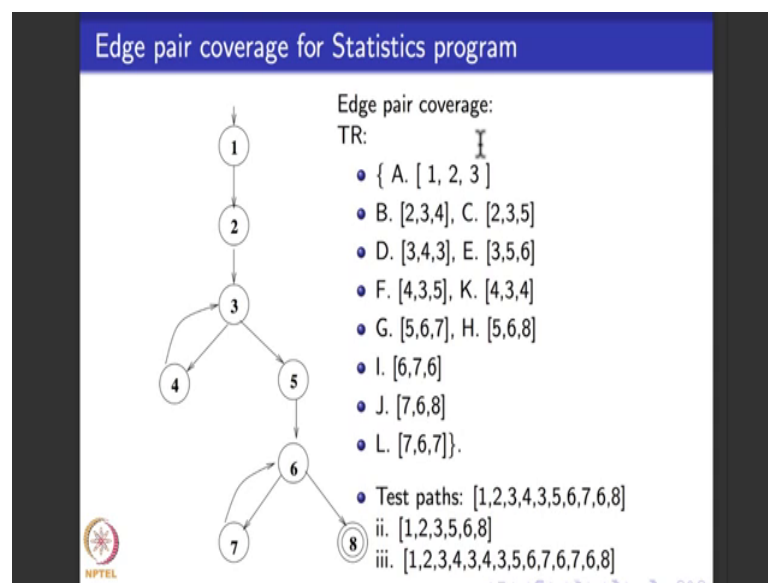
So, suppose I want to do edge coverage criteria for this graph. If you see you have taken CFG that was here and further removed all the labels retained only the core graph. Understanding is that this is the graph that corresponds to that java code that corresponds to this statistics program suppose I want to achieve edge coverage for this graph. What am I test requirements test requirement of TR says cover every edge. So, this is a set that contains the list of all the edges of this graph if you see as put all the edges in this set (1, 2), (2, 3), (3, 4), (4, 3) and so on.

And what is a test paths that will need this test requirement, it is fairly simple. I start from one I do (2, 3), (4, 3), I have covered these four edges. Then I take this branch, now 5 and then I do 6 and now I do not want to do 8 then I would have lost the edges 7, (6, 7) and (7, 6). So, I from 6, I got to 7, 7 to 6, 6 to 6 that is the path that I have to traced out here. I will just read out once again for clarity when I go from 1 to 2 to 3 and then move on to 4, come back to 3, come to 5, move to 6, come to 7, come back to 6 and then branch out to 8.

So, using one long test path that visits from the node 1, which is the initial node to the node 8, I have managed to achieve the test requirement of edge coverage. So, this is like a minimalistic test case, minimalistic because this is only one test path that is enough to test the coverage criteria of edge coverage for this graph. Of course, there nothing, no harm in you might say that this why one test path, I would write four test paths, I would write two different test paths like for example, I would write 1, 2, 3, 4, 3, 5, 6, 8 as one test path which are the edges that are not covered there these two edges.

And when you say fine I will write on other test path that cover those two edges, I could do 1, 2, 3, 5, 6, 7, 6, 8. So, my test case requirement of edge coverage is met through two test paths. That is also fine, anything is fine, but idea is the number of test paths we want to achieve for any kind of coverage criteria implicitly we also desired them to be as minimal possible. In this case, I could achieve it with just one test path, so I just wrote that, but there is nothing that says this is only option available we could do it with test paths that have two test paths, three test paths as long as you cover all the edges we are doing fine.

(Refer Slide Time: 32:52)



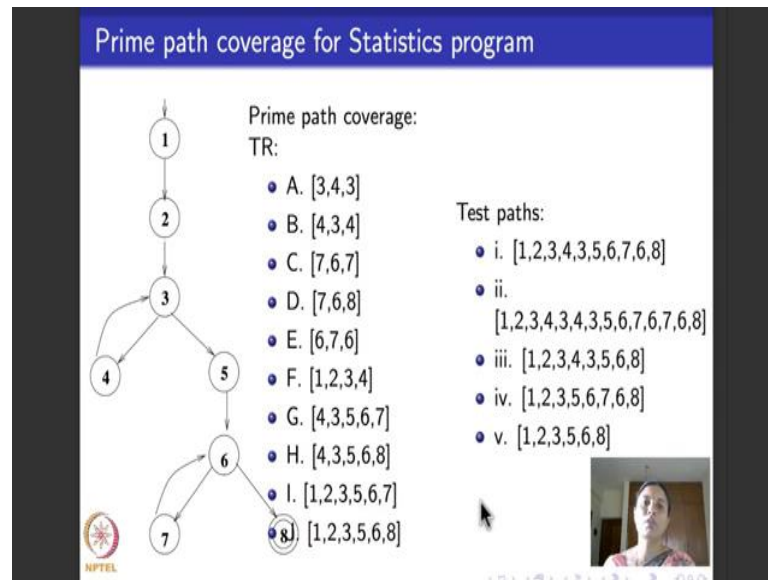
Now, let us say we want to do edge pair coverage for the same statistic program CFG. What is the test requirement for edge pair coverage, the TR for edge coverage is this set. I have written it such that I have given a label A, B, C, D, E F and so on for each test case. I will be using these labels in a few later lectures and I have also grouped them to

indicate the edge pair right like for example, 1, 2, 3 is this pair of edges (2, 3), I could do 4 or I could do 5. So, I have listed them together similarly from three I could do 3, 4, 3 or 3, 5, 6, so I have listed them together because both begin at 3. Similarly, the ones that begin at 4, I have listed them together, these two together and then the rest of them. So, this is my TR or test requirement for edge pair coverage which lists all paths of length two because it subsumes at coverage node coverage I am not listed those paths again and for this I will not be able to do it with just one test path.

So, this one test path is that long test path that we saw do before which is 1, 2, 3, 4, 5, 6, 7, 6, 8. So, this test path is needed. What does it miss out if you notice, which are the edge pairs that it misses out? Does it include for a example 5 6 8, it does not right because it does not come directly. If you see here, 5, 6, 7 comes 7, 6, 8 comes, 5, 6, 8 does not come and for now assume that I do not want to do side trips and detours. So, I have put another path that includes 5, 6, 8. So, I do 1, 2, 3, 5, 6, 8, so that is my second test path. And now we go back here and see what else as this missed. This has missed 4, 3, 4 also because if you see it is not there here it is defiantly not there here, because the paths does not if an come here. It is also missed 7, 6, 7 that is not in both these paths. So, I have another test path that includes both of them.

So, test paths always have to begin at 1 which is an initial node end at 8 which is a final node. So, it is fairly long path, because it has to go through the loop once again. So, I do 1, 2, 3, 4, 3, 4. That way I include this 4, 3, 4 and similarly for this one also I do 5, 6, 7, 6, 7, so that I can have 7, 6, 7 and then end it with 8. So, this is the test requirement or TR for edge pair coverage; and I need minimally three test paths to be able to achieve edge pair coverage for this graph.

(Refer Slide Time: 35:38)



So, we look at prime paths coverage. You remember I told you what are the algorithms to enumerate the test requirements for prime paths, so I run that algorithm on this graph, and this is the set of prime paths that I get. If you remember and go back to the slides that we had done then, you would see that I would have taken same example graph and we would have worked out the prime paths as these. So, I am just taking it. It so happens that this is the graph that comes the CFG for the statistics program, but these are the set of prime paths, so that is my TR and for meeting this TR, I need five different test paths right. So, this is how I do prime pair coverage. If you see the example, these five different test paths they will neatly execute the loop there are two loops in the graph, one for loop here and one for loop here, between these they will neatly skip the loop and execute the loop each in turn for both for loops. So, I have achieve loop coverage simplicity prime path coverage for this program.

So, what was what it we do in this module? We looked at how to draw control flow graphs corresponding to source code, and how to apply elementarily structural graph coverage criteria to test the control flow graph in turn the source code by using this criteria. In the next lecture, we will continue with source code. I will tell you what are the classical notions of testing related to source code, and how to they relate to the structural coverage criteria that we have seen so far.

Thank you.

Software Testing
Prof. Meenakshi D'souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

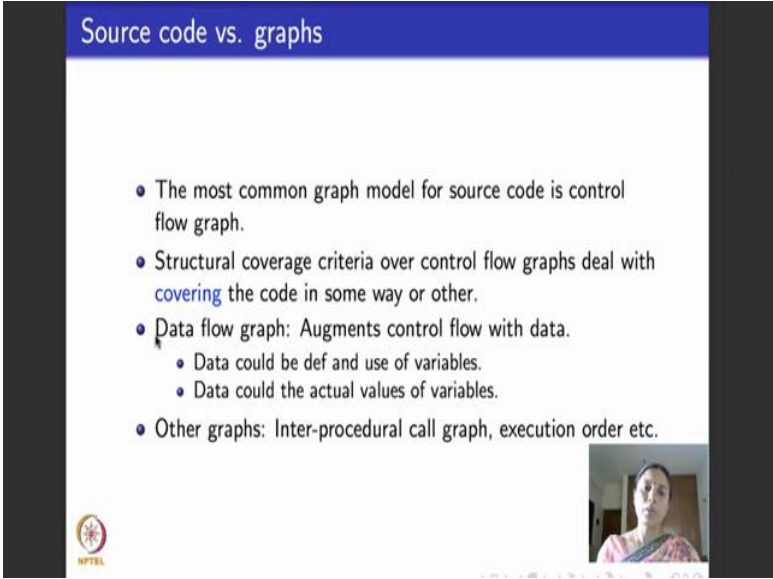
Lecture - 14
Testing Source Code: Classical Coverage Criteria

Hello again. We are going to look at the last lecture of week 3 today. So, the focus of this week has been two things: one is data flow coverage criteria on graphs and then what we did is we went back to structural coverage criteria. So, how to model source code as control flow graphs and how the various structural coverage criteria applied over them.

Today what we will do is that we will learn several graph coverage criteria--- structural and those that deal with data. And independently if you are familiar with little bit of software testing you would have heard several classical terms in testing branch coverage, cyclomatic complexity and so on. What we will do in this module is to see what are the terms that we have learnt till now, how are all of them related to classical terms in testing? In this process I will also point you to a couple of other good books in testing that you could use for general reading, part of the material of these courses is derived from those books.

So, for today's module we will not use the book by Paul Amman and Jeff Offutt, instead we will see testing as it is existed for the past four decades or so. What are the classical terms and how what we have done till now relates to these classical terms.

(Refer Slide Time: 01:26)



Source code vs. graphs

- The most common graph model for source code is control flow graph.
- Structural coverage criteria over control flow graphs deal with covering the code in some way or other.
- Data flow graph: Augments control flow with data.
 - Data could be def and use of variables.
 - Data could be the actual values of variables.
- Other graphs: Inter-procedural call graph, execution order etc.

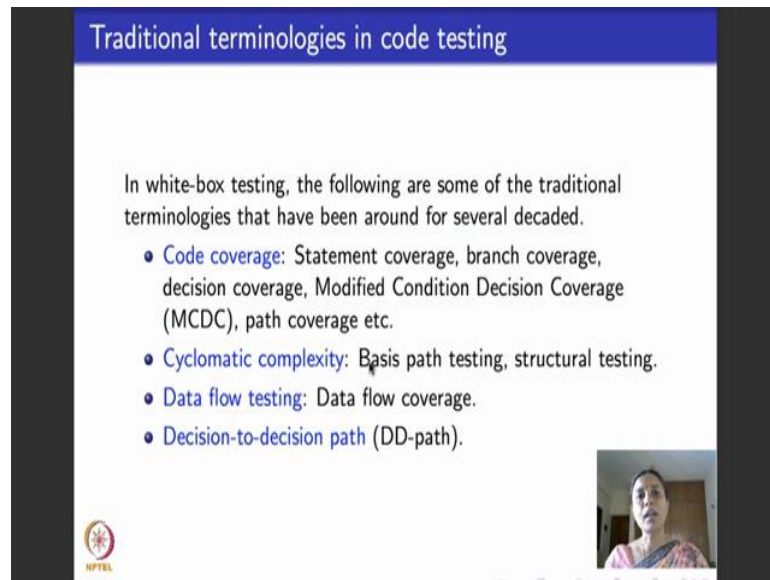
NPTEL

A small video inset in the bottom right corner shows a woman with dark hair, wearing a patterned top, speaking.

We know that the source code model can be written as a control flow graph as a graph model, what are the other models of the graph? The other common model of the graph is the data flow graph. The kind in data flow graph that we saw in this course took the control flow graph, augmented with definitions in uses. There is another kind of data flow graph that people use in program analysis that is, each node in the data flow graph is actually a triple representing the actual values of the various variables.

So, we will not really look at a data flow graphs for the scope of this lecture. There are several other graphs inter procedural call graphs execution order as I told you in the last lecture. What would we be doing today in this lecture is to go back in focus on the control flow graph model of a given piece of code and then understand what are the testing terminologies that have traditionally existed. Traditionally means you know somewhere from the mid seventies people have looked at these testing terminologies that we will be seeing today. So, for about four decades they have been around, very popular very well used, and will also understand how structural coverage criteria from this control flow graph relates to these classical terminologies that are always been used in test.

(Refer Slide Time: 02:43)



The slide is titled "Traditional terminologies in code testing" in a blue header. The main content is on a white background with black text. It starts with an introductory sentence: "In white-box testing, the following are some of the traditional terminologies that have been around for several decades." This is followed by a bulleted list of four items: "Code coverage" (listing statement, branch, decision, MCDC, and path coverage), "Cyclomatic complexity" (listing basis path testing and structural testing), "Data flow testing" (listing data flow coverage), and "Decision-to-decision path" (listing DD-path). In the bottom right corner, there is a small video inset showing a woman speaking. The NPTEL logo is in the bottom left corner.

Traditional terminologies in code testing

In white-box testing, the following are some of the traditional terminologies that have been around for several decades.

- **Code coverage:** Statement coverage, branch coverage, decision coverage, Modified Condition Decision Coverage (MCDC), path coverage etc.
- **Cyclomatic complexity:** Basis path testing, structural testing.
- **Data flow testing:** Data flow coverage.
- **Decision-to-decision path** (DD-path).

So, what are some of the classical terminologies that we will be looking at and relate them to graph coverage criteria? You might have heard some of these terms when it comes to white box testing. Ya, that is another thing, because we are looking at code and we are looking at structural coverage of code we are given white box testing right? So, in white box testing some of the popular terminologies that exist are what is called code coverage. Code coverage means it says you cover the code some way or the other in that you could do statement coverage; which says that you write test cases that will execute every statement in the code. You could do branch coverage which says you write test cases that we exercise each branch in the code. You could do decision coverage which will exercise test cases which will exercise each decision the various decisions could be those that reside in if statements as conditions in loops and so on.

Or you could do what is called modified condition decision coverage (MCDC) or you could do what is called path coverage. Things like decision coverage, MCDC and all are very popular terms that are used to test what are called safety critical software. So, we will see what these are and how they relate to the coverage criteria that we have looked at till now. Another popular term that you would have encountered while did testing was what is called Cyclomatic complexity right? Related to cyclomatic complexity people look at what is called basis path testing or structural testing.

Another popular term in testing is data flow testing that directly deals with data flow coverage mostly as we have dealt with in the course. And then, another popular term is what is called decision to decision abbreviated as DD-path testing. So, what we will do is that we will take each of these one after the other except for data flow testing. I will handle data flow testing separately as the first module of next week. So, we look at the other things which are coverage, cyclomatic complexity and DD-paths and see what they mean as far as their relationship to the kind of structural coverage criteria that we have seen till now.

(Refer Slide Time: 05:00)

Code coverage: Traditional vs. what we learnt

Assuming that we are working with the control flow graph model of code,

- Node coverage is the same as statement coverage.
- Edge coverage is the same as branch coverage.
- Prime path coverage is the same as loop coverage.

Decision coverage and MCDC will be covered when we do testing with logical predicates.

The slide also features a small video inset of a woman in the bottom right corner and an NPTEL logo in the bottom left corner.

As I told you we assume that we have a piece of code and we have derived the control flow graph of that code.

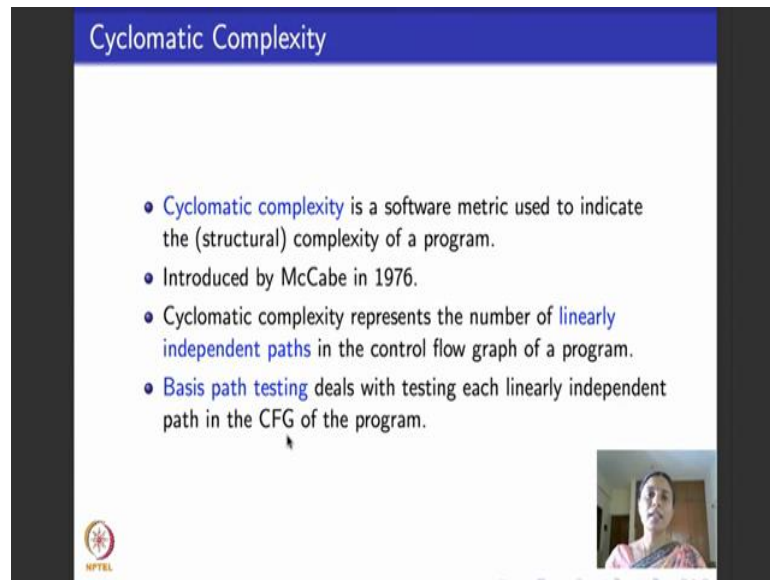
In the last module I took the statistics program example and showed you how to write the control flow graph piece by piece. In that module before that we saw pattern matching example and we saw the control flow graph and the data flow graph corresponding to the pattern matching example right. If you are working with control flow graph then what we saw as a node coverage criteria over graphs is the same as the statement coverage. Why is that so? Because if you see through examples the control flow graph, what is each node or vertex in the control flow graph correspond to? It corresponds to one statement or a set of statements that occur in sequence without branching in between them.

So, when I say my test requirement is node coverage, what I mean is execute every statement in the graph. So, what is popularly called as statement coverage is basically what we refer to as node coverage. Similarly edge coverage is the same as branch coverage. So, branch coverage says you take every branch in the graph right. So, ewhere there in branches come from? They come from edge statements, they come from loops, they come from switch case statements and they result in various edges that go out of the graph. So, when my test requirement is edge coverage then basically I am looking an branch coverage of the corresponding source code.

And then finally, loop coverage is another term. In white box testing they say you write test cases that will cover every loop it will skip the loop, it will execute the loop, once it will execute the loop for a few iterations within the maximum number of iterations allowed by the them. We all know that we saw prime path coverage. So, prime path coverage as a TR is basically the same as loop coverage in the underlying piece of code right. Now if you go back to the previous slide in the various coverage criteria we understood statement coverage, branch coverage, loop coverage as an extra thing, path coverage could be thought of as partly doing loop coverage, we said we won't explicitly look at complete path coverage right because several times it turns out to be an infeasible requirement. So, we did prime paths.

Today I will tell you few other kinds of path coverage criteria. What is left? Decision coverage and MCDC. Decision coverage and MCDC we will look at then we to testing with logic predicates as our models. So, when we finish graphs right in a few weeks from now, the next module that we will be beginning is to assume software artifacts as various kinds of logical predicates and we will design test cases based on those logical predicates. So, when we do that the coverage criteria that we see there would correspond to decision coverage and MCDC.

(Refer Slide Time: 08:00)



Cyclomatic Complexity

- Cyclomatic complexity is a software metric used to indicate the (structural) complexity of a program.
- Introduced by McCabe in 1976.
- Cyclomatic complexity represents the number of linearly independent paths in the control flow graph of a program.
- Basis path testing deals with testing each linearly independent path in the CFG of the program.

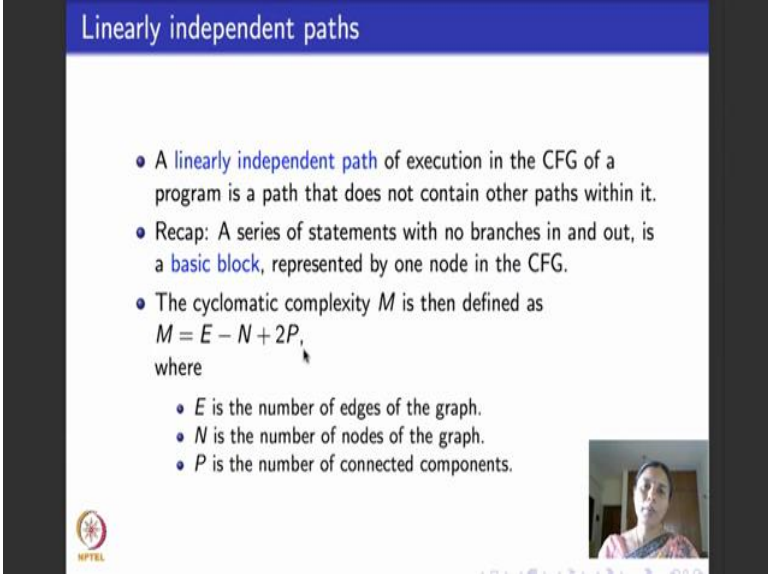
Moving on, the next popular testing term that you would have heard is what is called cyclomatic complexity. It is a pretty old term actually, it is more than 4 decades old, it was introduced by this person called McCabe in the year 1976.

So, what is cyclomatic complexity? It is a software metric; what is this software metric? Software metric is some kind of a measure about the piece of code that we have in hand. The most popular software metric is what is called lines of code and usually for fairly big software that are deployable in/come for commercial purposes, lines so there are several thousand lines of codes. So, this is a kilo lines of code a KLOC for every thousand lines of code. That is the most popular software metric. So, we say a large software has several KLOCs means several thousand lines of code.

Another popular metric that is used to measure how complex a software is, is what is called cyclomatic complexity. Here complexity is not measured in terms of the number of lines of code; it is measured in terms of the number of various branches that can occur in a software. So, what does cyclomatic complexity represent? It represents the number of linearly independent paths in the control flow graph for program. A testing that deals with cyclomatic complexity is what is called basis path testing. Basis path testing basically enumerates the number of different linearly independent paths and then tests with reference to the cyclomatic complexity of the program.

So, what can spend the next few minutes on is trying to understand how exactly to compute cyclomatic complexity given a control flow graph of a graph of a code, and what does basis path testing as a class of testing, a category of testing correspond to testing it with reference to the cyclomatic complexity measure.

(Refer Slide Time: 09:59)



The slide is titled "Linearly independent paths" in a blue header. It contains three bullet points: 1. A linearly independent path of execution in the CFG of a program is a path that does not contain other paths within it. 2. Recap: A series of statements with no branches in and out, is a basic block, represented by one node in the CFG. 3. The cyclomatic complexity M is then defined as $M = E - N + 2P$, where E is the number of edges of the graph, N is the number of nodes of the graph, and P is the number of connected components. There is a small video inset in the bottom right corner showing a person speaking, and an NPTEL logo in the bottom left corner.

- A linearly independent path of execution in the CFG of a program is a path that does not contain other paths within it.
- Recap: A series of statements with no branches in and out, is a basic block, represented by one node in the CFG.
- The cyclomatic complexity M is then defined as $M = E - N + 2P$, where
 - E is the number of edges of the graph.
 - N is the number of nodes of the graph.
 - P is the number of connected components.

So, with cyclomatic complexity as I told you deals with a number of linearly independent paths. So, we first need to understand what is a linearly independent path. What is a linearly independent path? It is a path that does not come as a sub path of any other path.

So, you take the control flow graph or the CFG of a program. There can be several paths in the graph. It is linearly independent path is similar to prime path. If you see prime path is a path that do not come as a sub path of any other simple path in the program. Linearly independent path is a path that do not come as a sub path of any other path in the program. So, except for the fact to the word simple is not there linearly independent paths are the same as prime paths, right. So, what we do is, how do we compute the cyclomatic complexity? We compute the cyclomatic complexity denoted as M , by using this formula.

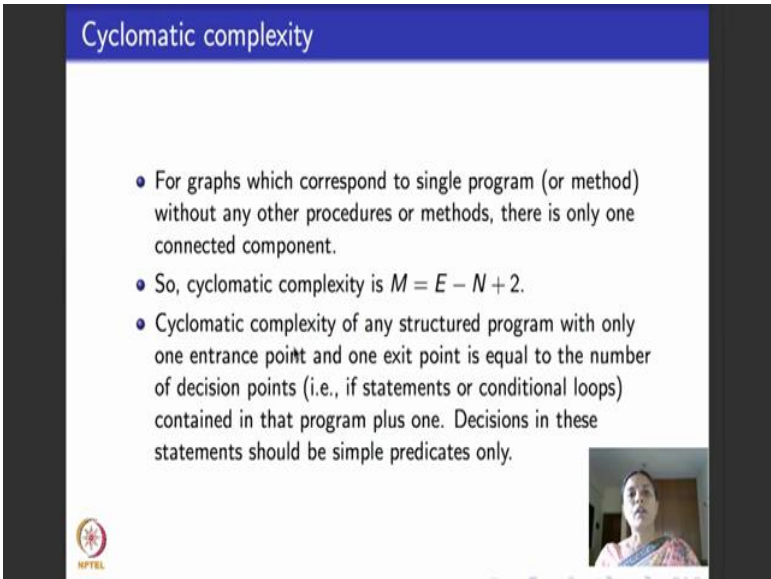
So, it says $M = E - N + 2P$. What is E ? E is the number of edges in the control flow graph, N is the number of nodes in the control flow graph what is P ? P is the number of connected components in the control flow graph. How do I obtain the control flow graph? When I calculate E N and P an make one important assumption about the control

flow graph. You remember when I told you that when you have a series a statements, let us say assignment statements or print statements one after the other you have the choice to keep one vertex or one node for each statement or you have the choice to collapse these series of statements as one vertex ,one individual vertex assuming that there is no branching in between them.

So such a series a statements is what we called a basic block; and when we do control flow graphs of several code fragments in the last module, I assumed one vertex for every basic block. For computing cyclomatic complexity that assumption turns to be very important, this formula will not be correct if you do not assign one vertex for one basic block. If you take the other approach where I assign one vertex for every individual statement, then this will not be the formula for cyclomatic complexity.

So, cyclomatic complexity assumes that the control flow graph is drawn in such a way that each node represents a basic block of statements, and different nodes exist only when there is a need to branch out of that node, only when there is a decision statement out of that node. Under that assumption that is how we built most control flow graphs throughout our lecture. So, for all those control flow graphs the formula to compute cyclomatic complexity is this, very easy to remember it is $E - N + 2 P$; where E is the number of edges N is the number of nodes, P is the number connected components.

(Refer Slide Time: 12:55)



The slide is titled "Cyclomatic complexity" in a blue header. It contains three bullet points: 1. For graphs which correspond to single program (or method) without any other procedures or methods, there is only one connected component. 2. So, cyclomatic complexity is $M = E - N + 2$. 3. Cyclomatic complexity of any structured program with only one entrance point and one exit point is equal to the number of decision points (i.e., if statements or conditional loops) contained in that program plus one. Decisions in these statements should be simple predicates only. In the bottom right corner, there is a small video inset showing a person speaking. The NPTEL logo is in the bottom left corner.

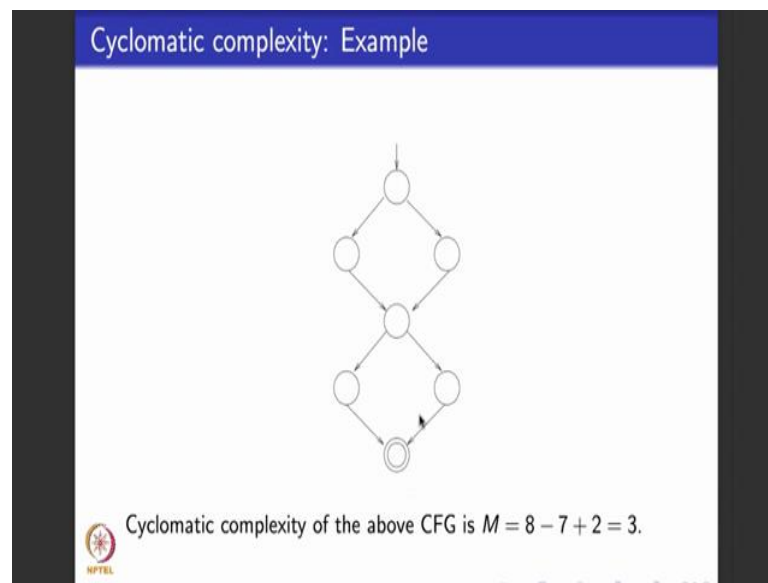
- For graphs which correspond to single program (or method) without any other procedures or methods, there is only one connected component.
- So, cyclomatic complexity is $M = E - N + 2$.
- Cyclomatic complexity of any structured program with only one entrance point and one exit point is equal to the number of decision points (i.e., if statements or conditional loops) contained in that program plus one. Decisions in these statements should be simple predicates only.

For graphs that correspond to a single program a single procedure or a single method assuming that there are no inter procedural calls that I model in my graph, then if you remember the control flow graph looks like one large connected component right. It does not look like several bits and pieces of disconnected components. If I consider that then this last term P is the number of connected components is basically one for a graph control flow graph that corresponds to one method right. So, the formula for cyclomatic complexity becomes simpler it is $E - N + 2$.

So, it so turns out that the cyclomatic complexity of a program with only one entry and one exit means with only one initial vertex and one final vertex, is the same as the number of decision points. What are decision points? Points where there is branching out if switch and so on. So, number of decision points contained in that program plus 1. So, what it says is that how do linearly independent paths come, I go through a series of statements at some point I branch out. When I branch out I have a choice I take the left branch or I take the right branch assuming that it is the branch of an if statement with an else part.

So, each loop adds to one linearly independent path, and what happens is the cyclomatic complexity of a program is basically the number of such branches plus 1, and it exactly gives the number of linearly independent paths in a program. And they say that usually a good indicator of good software that is easy to maintain and easy to handle should have cyclomatic complexity somewhere between 1 and 10. Cyclomatic complexity of 1 means the software has no branching. So, it may not be very useful to look at such software. So, cyclomatic complexity should be typically between 2 and 10. Any number less than 10 is considered to be a good cyclomatic complexity for a piece of software when it comes to maintaining and handling this software.

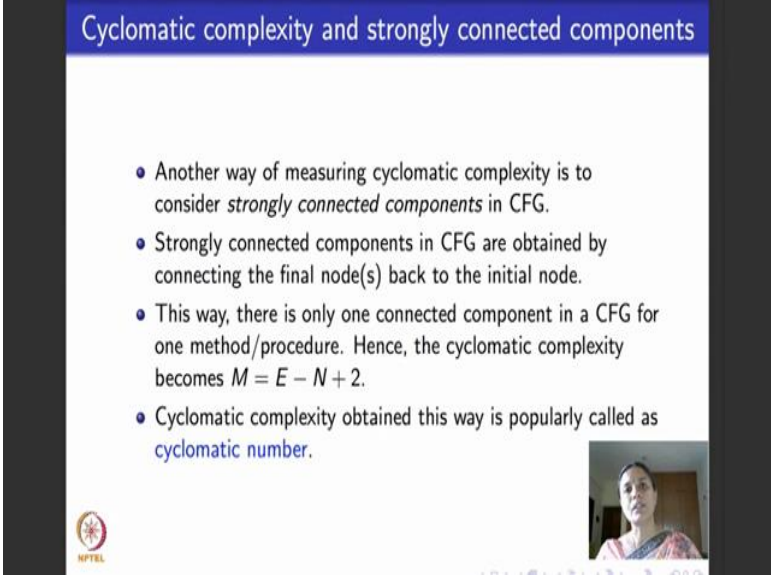
(Refer Slide Time: 14:55)



So, here is a small example. So, this software, this is the control flow graph corresponding to some piece of code. It has exactly one initial vertex marked here, it has one final vertex marked here and then it has two if statements. There is one branching here which branches out to this node and this node, there is one more branching here which branches out to this node and this node. So, there are two if statements; if you count the number of edges in this graph there will be eight edges, and there be seven vertices and this whole graph is like one connected component.

So, the formula, cyclomatic complexity turns out to be 3. If you see how many branches are there in this graph? There are two decision points one here and one here. So, what did we discuss here, we said the cyclomatic complexity is the number of decision points plus 1. That is true for this example; it is the number of decision points plus 1 which is 3.

(Refer Slide Time: 15:50)



The slide has a blue header with the text "Cyclomatic complexity and strongly connected components". Below the header, there is a list of four bullet points. The first bullet point states: "Another way of measuring cyclomatic complexity is to consider *strongly connected components* in CFG." The second bullet point states: "Strongly connected components in CFG are obtained by connecting the final node(s) back to the initial node." The third bullet point states: "This way, there is only one connected component in a CFG for one method/procedure. Hence, the cyclomatic complexity becomes $M = E - N + 2$." The fourth bullet point states: "Cyclomatic complexity obtained this way is popularly called as *cyclomatic number*." In the bottom right corner of the slide, there is a small video inset showing a woman speaking. In the bottom left corner, there is a small logo for NPTEL.

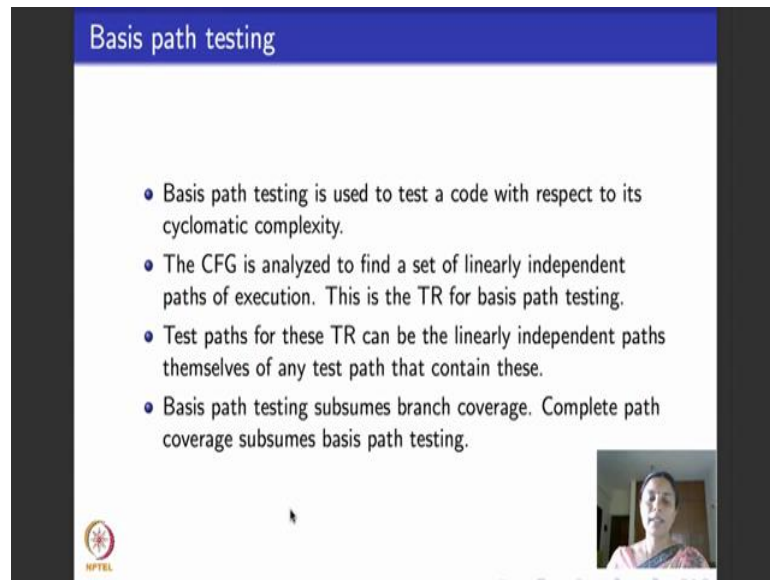
- Another way of measuring cyclomatic complexity is to consider *strongly connected components* in CFG.
- Strongly connected components in CFG are obtained by connecting the final node(s) back to the initial node.
- This way, there is only one connected component in a CFG for one method/procedure. Hence, the cyclomatic complexity becomes $M = E - N + 2$.
- Cyclomatic complexity obtained this way is popularly called as *cyclomatic number*.

So, that is another popular way of checking for cyclomatic complexity. What people do is the suppose this turns out to be the control flow graph of a particular program right. What they do is they put a dummy edge back from the finding vertex to it is initial vertex, like this, you trace assume that I am tracing an edge back from the final vertex, to it is initial vertex. After doing that the graph becomes one strongly connected component. You remember what strongly connected components are, we looked at them in the graph algorithms lecture. Strongly connected components are those components in which every pair a vertices are reachable from each other.

So, under that assumption cyclomatic complexity also turns out to be $E - N + 2$ basically because of the same reason, these exactly one connected component which actually happens to be strongly connected also. So, cyclomatic complexity becomes simpler, I do not have to compute connected components. It is usually easier this way because if you have something like this at the cost of adding a few extra edges, then you do not have to run algorithms that compute the number of strongly connected components as subroutines. So, directly count the number of edges, number of vertices, which I usually easy to do given a representation of the graph, and there you go, the cyclomatic complexity is easily measurable without calculating any strongly connected component.

So, usually this measure of cyclomatic complexity is preferred and it is popularly known as the cyclomatic number; some books and papers also call it the Betty number.

(Refer Slide Time: 17:23)



The slide is titled "Basis path testing" in a blue header. It contains a bulleted list of four points:

- Basis path testing is used to test a code with respect to its cyclomatic complexity.
- The CFG is analyzed to find a set of linearly independent paths of execution. This is the TR for basis path testing.
- Test paths for these TR can be the linearly independent paths themselves or any test path that contains these.
- Basis path testing subsumes branch coverage. Complete path coverage subsumes basis path testing.

In the bottom right corner, there is a small video inset showing a person speaking. The NPTEL logo is visible in the bottom left corner of the slide area.

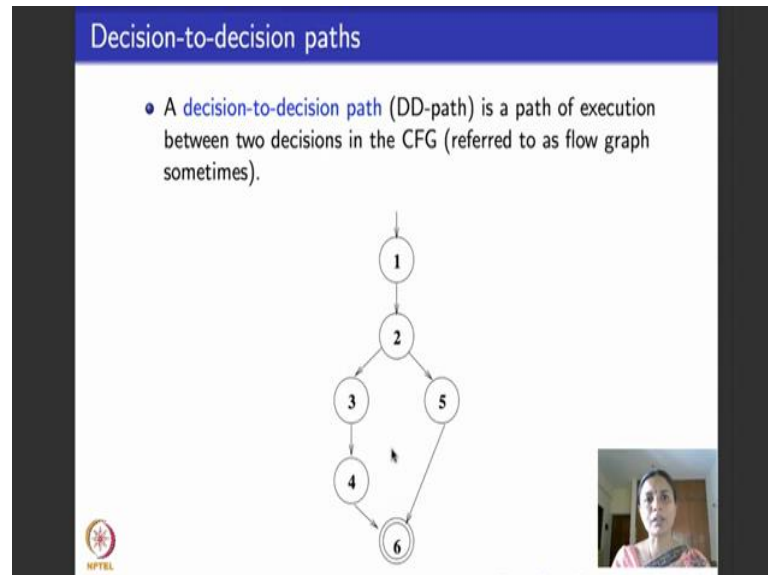
So, now let us look at basis path testing. As I told you basis path testing test the cyclomatic complexity of a piece of software, what it basically does is it enumerates all the linearly independent paths in the graph or in our terms, it enumerates all the prime paths in the graph, and for each such path, this is your TR, your test requirement. And any set of test paths that satisfy this TR would hold good for basis path test. Because we looked at prime paths and enumerate in prime path and not revisiting any algorithm that will enumerate linearly independent paths, it is very similar to this. So, we consider the same algorithm that we did for prime path as working for enumerating linearly independent paths and test it.

And like we did for prime paths coverage, basis path testing subsumes branch coverage or edge coverage, complete path coverage in turn subsumes basis path testing. If you are confused what I am trying to say is that basically linearly independent paths correspond to what we saw as prime paths, the only difference is simple path versus non simple paths. So, basis path testing is more or less the same as prime paths testing. So, how will I come up with a set of test requirements for basis paths? What are basis path testing test requirements? Enumerate all the linearly independent paths; it is the same as enumerating all the prime paths.

So, I will use the same algorithm that we saw for prime paths to enumerate all the linearly independent paths, that is my test requirement, and then I come up with the test

paths that satisfy the test requirement exactly in a way similar to the ones that we did for prime path test.

(Refer Slide Time: 19:05)



So, the last one that I wanted to deal with, another popular term that is been used in testing is what is called a DD-path or the decision to decision path. This is different from basis paths or linearly independent paths or from branches or from any of the kind of paths that we have seen throughout the span of this course. So, in short what is a DD-path? A DD-path is a path of execution between two decisions in the CFG. The books that talk about DD-paths usually call the control flow graph as the flow graph or a program flow graph, but if you are confused please remember that it is the same as what we call a CFG. At the end of these modules, this slides, this module I will point you to good references where you can read more information about cyclomatic complexity and about DD-paths.

For now we will just introduce what a DD-path is and say how we could go about testing for DD-paths. But so, here is the small control flow graph, its initial vertex is 1 final vertex is 6 and where are all the branches? There is a branching here, at 2 there is a possible choice of going to 3 or going to 5, and from 3 they you can go to 4 there is no choice there and if you see 6 there are two ways that you could use to come to 6, you could have come from this 2, 3, 4 and 6 or you could have come by using 2, 5 and 6.

So, I considered 6 also as having a decision point, as representing decision point in the graph, 2 as representing another decision point in the graph. Why because for both these kinds of vertices there is a choice about either coming in or going out of the vertex the choice in more than one way to come in or more than one way to go out. So, these are what are call decision points in the graph; and DD-path says you test, your test requirement of TR is a set of all paths between two decision points in the control flow graph.

(Refer Slide Time: 21:12)

Chains and DD-paths

- We will use the notion of chains to define DD-paths.
- A **chain** is a path in which
 - Initial and terminal vertices are distinct.
 - All the interior vertices have both in-degree and out-degree as 1.
- A **maximal chain** is a chain that is not a part of any other chain.

So, to see what a DD-path is in detail, we will need the notion of what is called a chain in a graph. You might have heard about the term chain when you looked at what are called partial orders. You typically see partial order is in a course with discrete math let us say, chains are what are call total orders there to the sake of simplicity and redefining chains here. So, what is the chain? A chain is a path which satisfies the following two conditions. The initial vertex and the terminal vertex of this path are distinct which means it is not a loop it is not a cycle, and then all the interior vertices, what are interior vertices? Interior vertices are those that are not the initial vertex and not the final vertex, everything else in between that comes in between. All the interior vertices have both in degree and out degree as one.

So, if you try to visualize how this chain will look like, it looks like one long path in the graph from an unique initial vertex to unique final vertex. And what is a maximal chain?

A maximal chain is a chain that is not a part of any other chain. Its somewhat like prime path right, a prime path is a path that is not a sub path of any other path, the maximal chain is chain does not a sub chain of any other chain.


(Refer Slide Time: 22:27)

DD-paths

A **DD-path** is a set of vertices in the CFG that satisfies one of the following conditions:

- It consists of a single vertex with in-degree 0 (initial vertex).
- It consists of a single vertex with out-degree 0 (terminal vertex).
- It consists of a single vertex with in-degree ≥ 2 or out-degree ≥ 2 (decision vertices).
- It consists of a single vertex with in-degree and out-degree as 1.
- It is a maximal chain of length 1.

The example graph has five DD-paths: [1], [2], [3,4], [5] and [6].



So, what is a decision to decision path a DD-path? A DD-path is basically a path of vertices in this CFG that satisfies any of the following five conditions. If it satisfies any of these five conditions we call them all as decision to decision paths.

So, let us look at the conditions one after the other. So, it could consist of a single vertex with in degree 0. If you think about it visualizing the CFG, which is that vertex that has in degree 0; what does it mean for a vertex do not in degree 0, there is nothing, no edge coming into that vertex. The only vertex that has in degree 0 is the initial vertex then it can consist of a single vertex without degree 0. Again you visualize our model of a control flow graph which is the vertex that has out degree 0, it is our final vertex or a terminal vertex, and then decision path can also consist of a single vertex with in degree greater than or equal to 2 or out degree greater than or equal to 2.

If we go back to the example graph that we had, vertex 2 is the vertex without degree as two, vertex 6 is a vertex with in degree as two. So, both vertices 2 and 6 satisfy this condition and both represent some kind of decisions that is what I was trying to explain a few minutes ago. Decision path can also consists of single vertex with in degree and out degree as both one, or a decision path could be a maximal chain of length one. So, just to

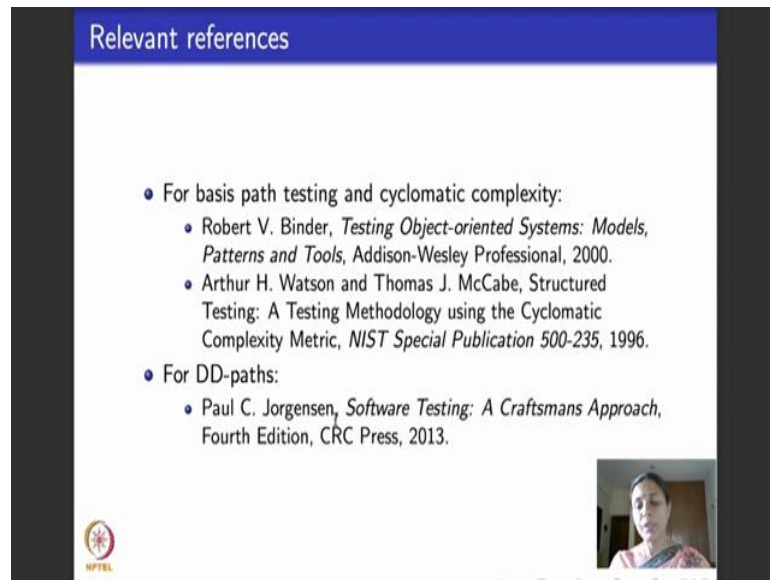
clarify, in case you find it confusing, what is the decision path? Basically it is a path that takes you from one decision to another decision. As I told you when we consider decisions, in for the sake of DD-paths, decisions could be of two kinds they could be of the classical kind like this, branch out or they could be of the kind where that is decision for branching in for coming in.

So, what it says is that DD-path let us you go from one decision to one decision. So, it clubs the vertices as all these things, it says it initial vertex is one separate DD-path final vertex is another separate DD-path then, all the decision vertices which are vertices with in degree greater than or equal to 2 or out degree greater than or equal to 2 are separate decision paths, and then, in between if I have a single vertex with in degree 1 and out degree 1 then that is another decision path, and then the other kind of decision paths could be chains of length 1.

So, if I apply this definition to this example graph, how many decision paths, DD-paths will I get? The initial vertex 1 is one DD-path final vertex 6 is one DD-path, the decision vertex 2 is a separate DD-path, the vertex 5 which has in degree 1 and out degree 1 is another DD-path, and 3, 4 is a chain of maximum length 1. So, this is another DD-path. So, totally for this graph there are five DD-paths: initial vertex, final vertex the decision point 2, vertex 5 which has in degree and out degree 1 and the maximum chain 3, 4 right. All of these satisfy one of these conditions that what is written here the example graph; that means so, as 5 DD-paths which have listed here.

So, we will not really see how to enumerate DD-paths and how to test them and all because they slightly how to scope for this course.

(Refer Slide Time: 26:21)



The slide is titled "Relevant references" in a blue header. It contains a bulleted list of references. In the bottom right corner, there is a small video inset showing a woman speaking. In the bottom left corner, there is a small logo for NPTEL.

Relevant references

- For basis path testing and cyclomatic complexity:
 - Robert V. Binder, *Testing Object-oriented Systems: Models, Patterns and Tools*, Addison-Wesley Professional, 2000.
 - Arthur H. Watson and Thomas J. McCabe, *Structured Testing: A Testing Methodology using the Cyclomatic Complexity Metric*, NIST Special Publication 500-235, 1996.
- For DD-paths:
 - Paul C. Jorgensen, *Software Testing: A Craftsmans Approach*, Fourth Edition, CRC Press, 2013.

But if you are interested in knowing more about it this is a very good book to learn more about DD-paths, it is a book called *Software Testing: A Craftsman approach*, by Jorgensen, and a recent addition is available. For getting to know about basis path testing and cyclomatic complexity, as I told you McCabe is the founder of Cyclomatic complexity he has an NIST report. NIST if you remember as National Institute of Standards and Technology. here is an NIST report which details how to find cyclomatic complexity, how to do basis path testing and how to use cyclomatic complexity for object oriented software for integration testing and so on it is a very good optical.

This another popular testing book the book by Robert Binder, which focuses on testing object oriented software also has an exhaustive discussion on cyclomatic complexity and basis path testing. So, if you are interested you could further read on from any of these books and feel free to ping me in the forum if you have any clarifications or questions that you would like a me to answer. But the main focus of this lecture was to help you understand that there are several classical terms that are present like the terms that we saw today, and see what they are and how they relate to whatever we have seen till now.

So, I hope you are not confused about why am I seeing these graph coverage criteria and how does it relate to cyclomatic complexity. And I hope this lecture would have helped you to understand and answer some of the questions that you had in your mind.

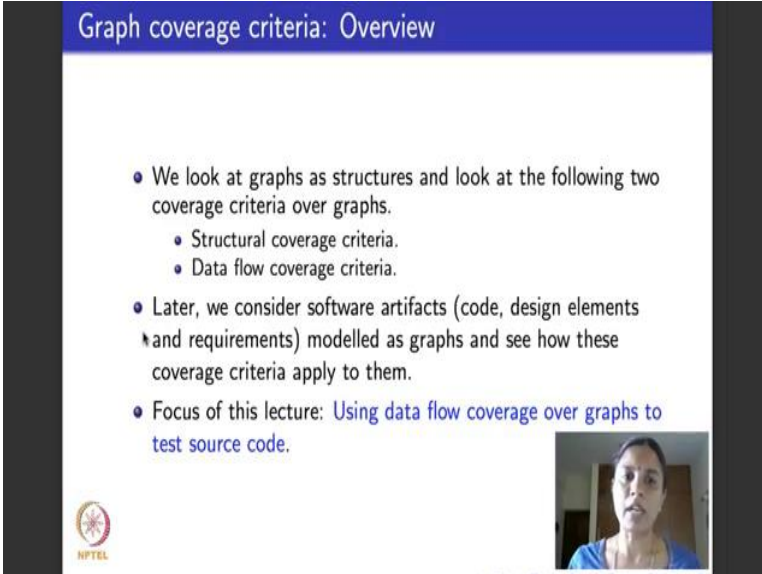
Thank you.

Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 15
Data flow graph coverage criteria: Applied to test code

Hello again, now we are in week 4. This is the first lecture of week 4. What are we going to do today? Today I will complete testing of source code using the graph models. If you remember last week we took source code then we looked at control flow graphs over source code and then we saw how the various structural coverage criteria applied for these. What we will be doing today is take source code again, but instead of looking at control flow criteria alone we will consider the CFG or the control flow graph which is augmented with defs and uses of data, and see how the data flow criteria that we had learnt about last week applies to testing of source code.

(Refer Slide Time: 00:56)



Graph coverage criteria: Overview

- We look at graphs as structures and look at the following two coverage criteria over graphs.
 - Structural coverage criteria.
 - Data flow coverage criteria.
- Later, we consider software artifacts (code, design elements and requirements) modelled as graphs and see how these coverage criteria apply to them.
- Focus of this lecture: Using data flow coverage over graphs to test source code.

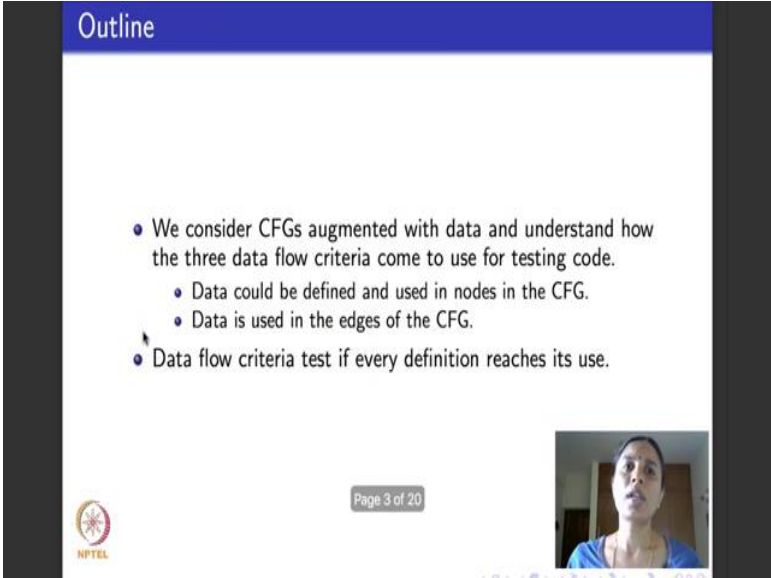
The slide also features the NPTEL logo in the bottom left corner and a small video inset in the bottom right corner showing Prof. Meenakshi D'Souza.

This is what we have done till now it is a summary of what we have done. So, we have looked at graphs the structures and we have learnt about 2 kinds of coverage criteria: structural coverage criteria and data flow coverage criteria. We learnt various coverage criteria, see how they subsumed, then what we saw last week was we took source code, we learnt how to draw CFG modules for each bits of design construction, source code and then we took an example, we took the whole CFG for that example and learnt how to

apply structural coverage criteria to test that source code by applying the coverage criterion the CFG.

Today what we will do is we will take the source code, take the CFG augment the CFG with defs and uses and see how to apply the 3 data flow criteria that we learnt on those things to test the source code.

(Refer Slide Time: 01:46)



Outline

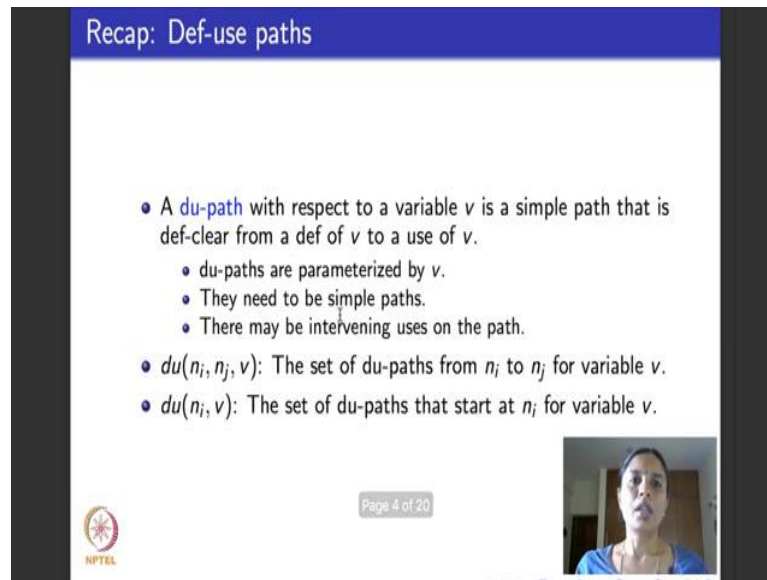
- We consider CFGs augmented with data and understand how the three data flow criteria come to use for testing code.
 - Data could be defined and used in nodes in the CFG.
 - Data is used in the edges of the CFG.
- Data flow criteria test if every definition reaches its use.

NPTEL

Page 3 of 20

So, to start with I will recap data flow coverage criteria, and then like we did in the previous example, we will take one example code, draw a CFG, look at the defs and uses and see what applying data flow coverage criteria to them will mean. So, as I told you we look at CFGs augmented with data, data is augmented as definitions and uses definitions and uses. Definitions and uses occur in the nodes of the CFG, there are no typically definitions in the edges of this CFG when it comes to modeling code; definitions come only a nodes edges have uses in them. So, what are data flow criteria's goal? It is to be able to write test paths that check if definition reaches is used in 1 way or the other, right.

(Refer Slide Time: 02:31)

A screenshot of a presentation slide titled "Recap: Def-use paths". The slide contains a bulleted list of definitions and notations for def-use paths. In the bottom right corner, there is a small video feed of a person speaking. The NPTEL logo is in the bottom left, and a "Page 4 of 20" indicator is in the bottom center.

Recap: Def-use paths

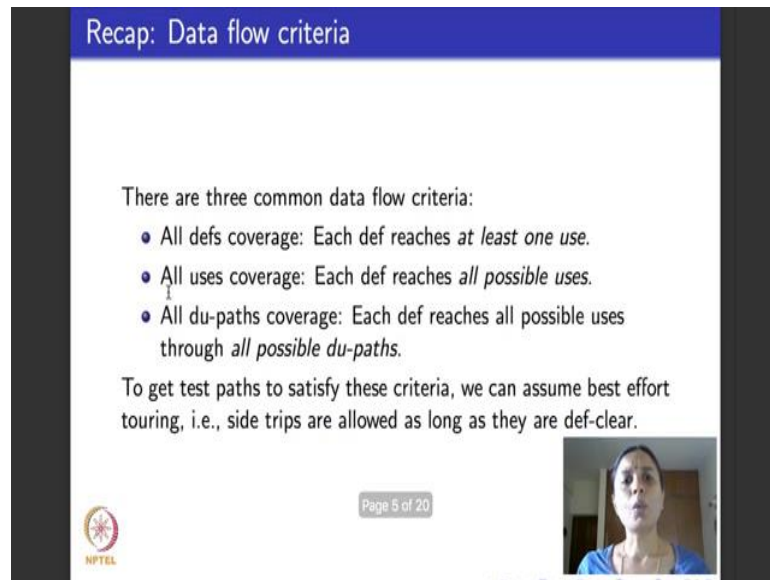
- A **du-path** with respect to a variable v is a simple path that is def-clear from a def of v to a use of v .
 - du-paths are parameterized by v .
 - They need to be simple paths.
 - There may be intervening uses on the path.
- $du(n_i, n_j, v)$: The set of du-paths from n_i to n_j for variable v .
- $du(n_i, v)$: The set of du-paths that start at n_i for variable v .

Page 4 of 20

So, we recap a few basic concepts about data flow criteria that we learned from the last weeks lectures, we learnt what is called a def use path abbreviated as d u-path; d u-path are with always reference to a fixed variable v .

What sort of paths they are? The first thing to notice that they are simple paths they do not have any cycles, and they trump from a node that contains a definition of v to a node or an edge that contains a use of v . What we insist is that in between these 2 definition and the corresponding use, there are no further definitions of v in all the intermediate vertices along the simple paths. So, this is what is written here we have parameterized by the variable v , they need to be simple, and that there need to be no intermediate definitions, but they could be intervening uses, we really do not worry about that, and for a variable v if there is a du-path going from n_i to n_j we write it as $du(n_i, n_j, v)$ for a variable v the set of d u paths beginning at the node n_i is written as $du(n_i, v)$.

(Refer Slide Time: 03:40)

A presentation slide titled "Recap: Data flow criteria" with a blue header. The slide lists three common data flow criteria: All defs coverage, All uses coverage, and All du-paths coverage. It also mentions that to get test paths to satisfy these criteria, we can assume best effort touring, i.e., side trips are allowed as long as they are def-clear. The slide includes an NPTEL logo in the bottom left, a "Page 5 of 20" indicator in the bottom center, and a small video feed of a person in the bottom right.

Recap: Data flow criteria

There are three common data flow criteria:

- All defs coverage: Each def reaches *at least one use*.
- All uses coverage: Each def reaches *all possible uses*.
- All du-paths coverage: Each def reaches all possible uses through *all possible du-paths*.

To get test paths to satisfy these criteria, we can assume best effort touring, i.e., side trips are allowed as long as they are def-clear.

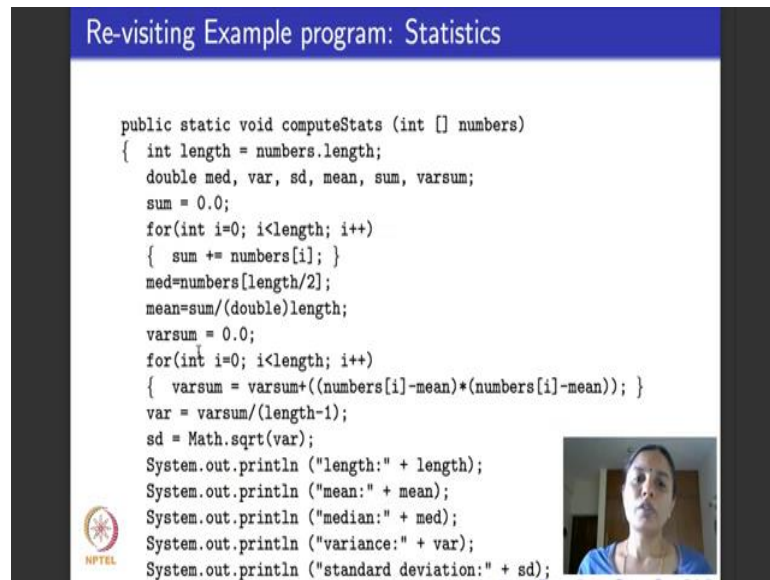
NPTEL

Page 5 of 20

What are the 3 dataflow criteria that we learnt? Not as many a number of structural coverage criteria only 3 in number. So, if you remember the 3 criteria deal with how the definition goes to its use. The first one was called all defs coverage, it basically insisted that each definition reaches at least one use; it basically insists that the variable is never defined and not used at all. The second criteria says all uses coverage; it insists that each definitions reaches all possible uses. The third criteria called all du-paths coverage says that each definition reaches all possible uses using all possible different paths that take the definition tools use, and another thing to remember is that when we consider these 3 data flow criteria what are called test requirements, right.

So, when we consider test paths to test these test requirements, we assume that the test paths can be taken with what is called best effort touring. So, which means you are allowed side trips and detours as long as it remains to be a test path and you can use the test paths to satisfy data flow coverage criteria. Oe extra requirement that we have about this side trips that could come in best effort touring, is that this side trip should still be definition clear that is always a condition that we insist.

(Refer Slide Time: 05:09)



Re-visiting Example program: Statistics

```
public static void computeStats (int [] numbers)
{
    int length = numbers.length;
    double med, var, sd, mean, sum, varsum;
    sum = 0.0;
    for(int i=0; i<length; i++)
    {
        sum += numbers[i];
    }
    med=numbers[length/2];
    mean=sum/(double)length;
    varsum = 0.0;
    for(int i=0; i<length; i++)
    {
        varsum = varsum+((numbers[i]-mean)*(numbers[i]-mean));
    }
    var = varsum/(length-1);
    sd = Math.sqrt(var);
    System.out.println ("length:" + length);
    System.out.println ("mean:" + mean);
    System.out.println ("median:" + med);
    System.out.println ("variance:" + var);
    System.out.println ("standard deviation:" + sd);
}
```

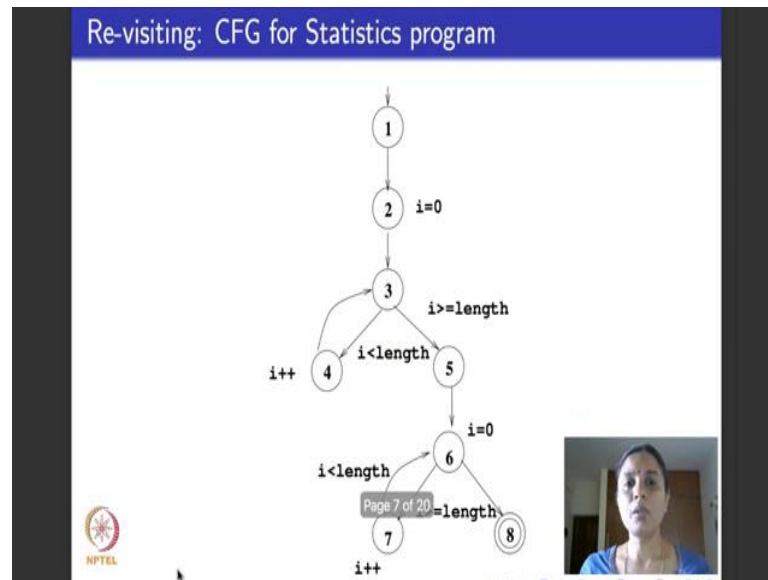
The slide also features an NPTEL logo on the bottom left and a small video inset of a woman speaking on the bottom right.

So, if you remember we had looked at the statistics examples when we had seen the control flow graph and how to draw it, I am recapping the same example this is the code for the same statistics program.

We will quickly recap what it does, it takes an array called numbers and then it is idea is to compute various statistical parameters about this array. What are the various statistical parameters that it computes? It computes the median, variance, standard deviation means sum and the variance and how does it go about doing? It is very simple this formulae as straight forward initializes the sum to 0, computes the sum in a for loop, then it computes the median mean and then it initializes varsum to 0 uses another for loop to compute the varsum and then it computes the variance and standard deviation, and it prints all the values that it is computed, right.

So, this is how the program looks like. So, if you remember the control flow structure of the program, there are 2 for loops one here and one here and the whole set of assignment statements before the first for loop, 3 statements in between the 2 for loops, and a few statements and belong with printouts after the third for loops.

(Refer Slide Time: 06:18)



So, here was the control flow graph for the statistics program that we had drawn last time. Node 1 is where we begin and the node 2 initializes the for loop, node 3 is this dummy node corresponding to the CFG of the first for loop, this (3, 4) structure here enters and executes the first for loop.

This 3 to 5 means I have finished computing the sum here in the first for loop, and move on to compute the median, mean, variance sum and then go on to the second for loop. So, that is this path--- 3 to 5, I do all the other initializations the other computation, sorry, median and other things here at node 5, and then I move on to node 6 where I begin the second for loop. This cycle of length 2 between 6 and 7 executes the second for loop, and when I take from 6 to 8 I finished my execution, 8 is the node where I go ahead compute this var, standard deviation and do all these printouts.

So, in this CFG that we have drawn here corresponding to the statistics program, we have given labels corresponding only to the length. I have retained the same structure that we used it in the last module, we have not given all other labels just to reduce clutter. But I hope you remember what labels node 1. The statements that label node 1 are all these statements before the beginning of the for loop, 2 begins the for loop 3 to 4 executes the for loop 3 to 5 comes out of the for loop, at node 5 we do all these computations--- median, mean, initialized varsum; node 6 begins the second for loop the

cycle 6 to 7 executes the second for loop, 6 to 8 exits the for loop, at node 8 we compute var, standard deviation and do all these printf's, right. So, that is the CFG.

(Refer Slide Time: 08:23)

Statistics program: Definitions and uses at nodes

Node	Defs	Uses
1	{numbers,sum,length}	{numbers}
2	{i}	
3		
4	{sum,i}	{numbers,i,sum}
5	{med,mean,varsum,i}	{numbers,length,sum}
6		
7	{varsum,i}	{varsum,numbers,i,mean}
8	{var,sd}	{varsum,length,var,mean,med,sd}

Page 8 of 20

So, if you look at the CFG it depicts plane control flow graph. My goal is to be able to augment the CFG with data information specifically with information about definitions and uses of data, that is what we are going to be able to do it. Again what I have done is have drawn tables, where I have taken nodes from the CFG we will go back to the control flow graph, if you see the nodes are numbered 1, 2, 3, 4 and so on up to 8. At each node certain variables are defined and certain variables are used. So, this table lists which are the variables that are defined at each of the nodes and which are the variables that are used at each of the nodes.

So, if you see it says at node 1, three variables are defined. What are the 3 variables, numbers sum and length. So, let us go back and see how they are defined at node 1. So, this is node 1 in the CFG, it marks the node where all statements that happen before the execution of the first for loop are. So, we go back one slide which are the statements these are the statements there is this length is numbers.length by compute the length of the array, I do all these initial the declarations and then I initialize. So, it says node 1 represents that and the variables that are defined at node 1 are these array numbers which is taken as input, sum which was initialized to 0 and length which computes the length of the array.

So, that is what these statements correspond to numbers which was taken as input, length which initially which computed the length of the array, and sum which was initialized to 0.

So, it says at node 1, which corresponds to those statements these are the definitions. What are the uses at node 1? It says the array numbers is used at node 1. Why is that so? If you go back to the statement, we have this statement right which says `length = numbers.length`. So, it says you compute the length of this array numbers and set it to the variable length.

So, the numbers as an input, as a variable, which is an array data type is also used at node 1. Node 2 is very simple it initializes the index corresponding to the for loop. So, the only variable that is defined at node 2 is `i`, nothing is used at node 2. We go back to the CFG see node 3, you remember node 3 corresponds to the beginning of the for loop and in the module where we discussed how to draw control flow graphs corresponding to loops, I told you that CFG is corresponding to loops will have a dummy nodes. Dummy node in the sense the node that is meant for the loop to come back to; 3 such a node because there is a dummy node corresponding to the for loop there are no definitions and no uses at node 3. So, this whole row is left blank.

I move on. At node 4 what happens the for loop gets executed. Let us go back to the code and see which is the statement that gets executed in the for loop, that is this. So, it a just take the variable sum, add the value of the next number that you find in the array numbers to the sum, and keep repeating it for the entire length of the array. So, the definitions at sum at node 4 are sum, because it is set back to sum and `i` and what are the uses? Uses are the array numbers, `i` and sum again. Is that clear? At node 5 what happens if you go back to the CFG node 5 we are out to the for loop, which means we finished going through this array compute that the sum of all the numbers that we found in the array, we are out of this for loop we are doing these three statements we are computing the median, we are computing the mean, and we are computing, we are initializing varsum.

So, the defs at node 5 are median, mean, varsum and `i`. Why is `i` there? `i` is there because I still came out at node 5 when `i` exceeded the number of iterations of the for loops, `i` still gets defined at node 5, the last increment of `i` happens at node 5. So, the defs at node 5

are all these, the uses at node 5 are numbers length and sum; because these statements use those variables to be able to compute the mean, median and initialize varsum. So, the defs and uses at node 5 I hope are clear. Node 6 if you go back, again is a dummy node that corresponds to the loop.


So, I that is no defs uses for node 6, I move on like that; node 7 corresponds to the execution of the second for loop. So, the defs at node 7 are these, the uses at node 7 are these. If you remember node 7 was inside this for loop where this is computed, varsum is computed. What do I use here? I use the variable mean, I use varsum which I initialize to 0 here, and I use the array number. What do I use them for? I use them to define varsum once again. So, that is what the table indicates. I use varsum numbers I and mean, and I define varsum and I also increment I in node for a for loops that also comes as definitions. At node 8 I am out of this for loop and I do all these statements.

So, the Defs and uses at node 8 are like this. I define variance and standard deviation and I print out all these values. So, because I print out all these values at node 8, they are all come as uses for node 8. So, is it clear please, how this table has arrived at? I take the CFG list all the vertices of the CFG; per vertex, I go back and see what are the definitions, what are the uses.


(Refer Slide Time: 14:11)

Statistics program: Uses at edges

Edge	Use
(1,2)	
(2,3)	
(3,4)	{i,length}
(4,3)	
(3,5)	{i,length}
(5,6)	
(6,7)	{i,length}
(7,6)	
(6,8)	{i,length}



Page 9 of 20



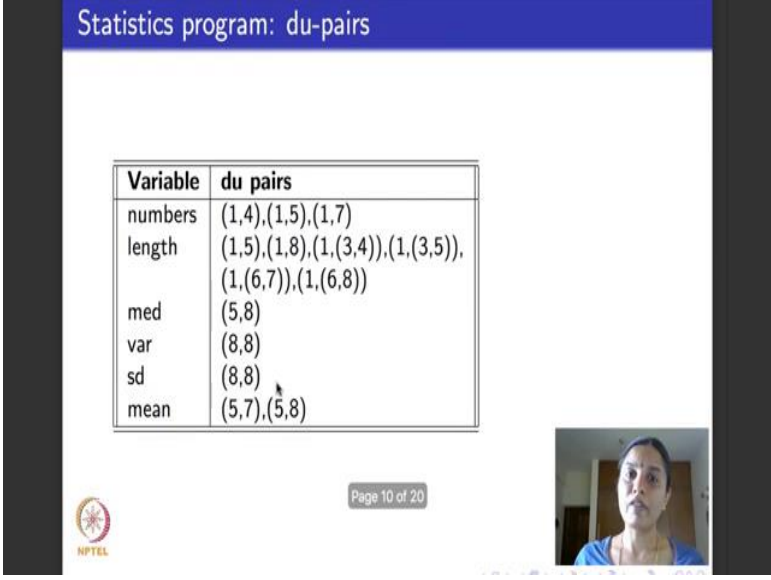
So, what we do now is repeat this exercise for the edges of the CFG. Remember when we talked about edges of a control flow graph there are no definitions associated with edges of a control flow graph corresponding to code.

So, typically it will only be uses and again where do uses come? Uses come only in those edges that involves checking of conditions. So, if you go back and then see the CFG the uses, which are the edges that correspond to the users will be very clear. The edge (3, 5) means that I am exiting the for loop. So, this condition holds the invariant of the for loop is no longer true, the edge (3, 4) means I am executing the next iteration of the for loop. So, this condition holds. So, these correspond to uses at these 2 edges. Similarly we uses this at these 2 edges (6, 8) and (6, 7) are these, that is what is written here and all other edges there are no uses, so they are left blank.

So, the uses at edge (3, 4) are i and length, the uses at edge (3, 5) are i and length, the uses at edge (6, 7) and (6, 8) which correspond to the execution of the second for loop are again i and length. Please remember that in this code we have used the same variable i to represent the first for loop and second for loop and when I document defs and uses, I again reuse the same variable like; it really does not matter, but if you find it very confusing you could call the index of the second for loop using another variable, you could call it as a variable j and then track defs and uses for j. It will still be the same right there is no harm in using the same variable again, right.

So, what have we done till now we have taken the control flow graph, taken each node in the control flow graph written down what which variables are defined at a node which variables are used at node. And then, we took each edge in the control flow graph, and talked about are there any variables that are used at these edges and if there are what are they, that is this table. Now the next step after we have documented the definitions and uses is to compute definition use pairs right. So, now, if you remember du-pairs are always parameterized by a variable. So, per variable, I document to the various def use pairs or du-pairs.

(Refer Slide Time: 16:21)




Statistics program: du-pairs

Variable	du pairs
numbers	(1,4),(1,5),(1,7)
length	(1,5),(1,8),(1,(3,4)),(1,(3,5)), (1,(6,7)),(1,(6,8))
med	(5,8)
var	(8,8)
sd	(8,8)
mean	(5,7),(5,8)

Page 10 of 20

NPTEL



So, for the variable numbers let me go back, numbers is defined at node 1 and where it is used ? It is used at node 4, it is used at node 5, and it is used at node 7, is it clear. It is also used at node 1 right so, I write that in this table. For the variable numbers it is defined at node 1 used at node 4, defined a node 1 used at node 5, defined a node 1 used at node 7. So, you might wonder it is also defined and used at node 1, why have I not listed it in this table? I have not listed it in this table because that will violate the condition that from node 1 to node 1 the path is def clear right because it is defined and used to the same node.

So, when that happens we typically do not list it as a du-path; because it violates the condition of the path being simple and it violates the condition of the path being def clear I list everything else and I do is exercise for each of the variables. So, I repeat this exercise for length, for median, variance, standard deviation mean, sum, varsum and I right this data spread across two slides. So, what is it for length? Let us go back, if you see length is defined at node 1, you see here the length is defined at node 1 whereas, length used again it is used a node 5, it is used a node 8, and when it comes to uses we have to look at edges also it is used at node edge (3, 4), it is used at edge (3, 5), (6, 7) and (6, 8) that is what is listed here.

Its defined at node 1, used at 5, defined at 1 used at 8 and defined at 1 and used at all these various edges. So, please read this as defined at 1 used at the edge (3, 4), right I go



on repeating this exercise the variable, median this defined at node 5 used at node 8 that is what is given here similarly for var, standard deviation, mean, sum, varsum.

(Refer Slide Time: 18:48)

Statistics program: du-pairs, contd.

Variable	du pairs
sum	(1,4),(1,5),(4,4),(4,5)
varsum	(5,7),(5,8),(7,7),(7,8)
i	(2,4),(2,(3,4)),(2,(3,5)),(2,7),(2,(6,7)),(2,(6,8)) (4,4),(4,(3,4)),(4,(3,5)),(4,7),(4,(6,7)),(4,(6,8)) (5,7),(5,(6,7)),(5,(6,8)) (7,7),(7,(6,7)),(7,(6,8))

Page 11 of 20

And for i, if you see the list of the d u pairs is quite vast, because I is the index of the variable for loop and it comes in two for loops in the code right.

And if you see here in these tables also in the use of edges this i all over, in the uses, in the defs of nodes there is i here there is i here correct. So, the du-pairs for the variable i is quite a bit. This one that begins at 2 talks about i being used in the first for loop, (2, 4) to an edge (3, 4), to an edge (3, 5), to a 7, to an edge (6, 7) this one talks about it is i being used in second for loop right. So, the du-pairs for i is a large set right.

So, is it clear what we have done till now, we took the CFG augmented the CFG, with Defs and uses augment at each node of the CFG with definition and use augment at each edge of the CFG with uses; then which just repopulated the table we said we will take one variable at a time and used the pairs where the variable is defined and used. The definition of a variable typically comes only from the node; the use of variable could come from a node or from an edge. So, this is the set of all the du-pairs corresponding to each of the variables in the code these two tables depict that.

(Refer Slide Time: 20:22)

Statistics program: DU paths		
Variable	DU pairs	DU paths
number	(1,4),(1,5),(1,7)	[1,2,3,4],[1,2,3,5],[1,2,3,5,6,7]
length	(1,5),(1,8)	[1,2,3,5],[1,2,3,5,6,8]
	(1,(3,4)),(1,(3,5))	[1,2,3,4],[1,2,3,5]
	(1,(6,7)),(1,(6,8))	[1,2,3,5,6,7],[1,2,3,5,6,8]
med	(5,8)	[5,6,8]
var	(8,8)	No path needed
sd	(8,8)	No path needed
mean	(1,4),(1,5)	[1,2,3,4],[1,2,3,5]
	(4,4),(4,5)	[4,3,4],[4,3,5]

Now, after we have had d u pairs we are ready to define definition use paths. So, we take the same data per variable which are the du-pairs, and then say; which are the paths that correspond to this du-pairs. So, take the variable number it is defined at 1 used at 4, and the paths that takes it from the definition at 1 and to the use at 4 is this path: 1 to 2 to 3 to 4. Similarly the variable number is defined at 1 used at 5, and the path that takes it from the definition at 1 to the use at 5 is this path 1, 2, 3, 5. Take the variable number--- defined at 1 used at 7 and this is the path 1, 2, 3, 5, 6, 7 that takes the definition of the variable number from it is definition at 1 to it use at 7.

Similarly, let us say for from median it is defined at 5, and used at 8. So, the paths that takes it from it is definition at 5 to it is use at 8 is the path 5, 6, 8; for var and sd is defined and you study it I as I told you I need not have listed this at all even as I do you pair. So, I do not list du-paths because they are not def free and they are not simple paths I ignore them and get going. So, I go on listing for all the variables in the program this are the pairs this is the paths right.

Now, if you see this, concentrate on this column corresponding to du-paths, you will realize that several paths repeat right. So, if you take this 1, 2, 3, 5; 1, 2, 3, 5 comes here again 1, 2, 3, 5 comes here again, here again similarly the path 1, 2, 3, 4 here once twice thrice so, on right.

(Refer Slide Time: 22:15)

Statistics program: DU paths		
Variable	DU pairs	DU paths
mean	(5,7),(5,8)	[5,6,7],[5,6,8]
varsum	(5,7),(5,8)	[5,6,7],5,6,8]
	(7,7),(7,8)	[7,6,7],[7,6,8]
i	(2,4),(2,(3,4)),(2,(3,5))	[2,3,4],[2,3,4],[2,3,5]
	(4,4),(4,(3,4)),(4,(3,5))	[4,3,4],[4,3,4],[4,3,5]
	(5,7),(5,(6,7)),(5,(6,8))	[5,6,7],[5,6,7],[5,6,8]
	(7,7),(7,(6,7)),(7,(6,8))	[7,6,7],[7,6,7],[7,6,8]

And if you go for other variables also if you see the path 5, 6, 7 comes here, here, here and here similarly 5, 6, 8 gets repeated, 7, 6, 7 is repeated, there are several repetitions. So, if you count the du-paths across the last column of these two slides, you will realize that there are totally 38 of them, but lot of them repeat. So, there are only 12 that are unique, right.

(Refer Slide Time: 22:33)

Statistics program: Du-paths without duplicates	
There are 38 du-paths for Stats, but only 12 of them are unique.	
• Paths that skip a loop:	
• Four paths: [1,2,3,5], [1,2,3,5,6,8], [2,3,5], [5,6,8]	
• Paths that require at least one iteration of a loop:	
• Six paths: [1,2,3,4], [1,2,3,4,6,7], [4,3,4], [7,6,7], [4,3,5], [7,6,8]	
• Paths that require at least two iterations of a loop:	
• Two paths: [4,3,4], [7,6,7]	

So, these are the 12 unique du-paths, I have grouped them into three sets here. So, the first set which consists of 4 paths a paths that skip one of the loops in the program. Let us

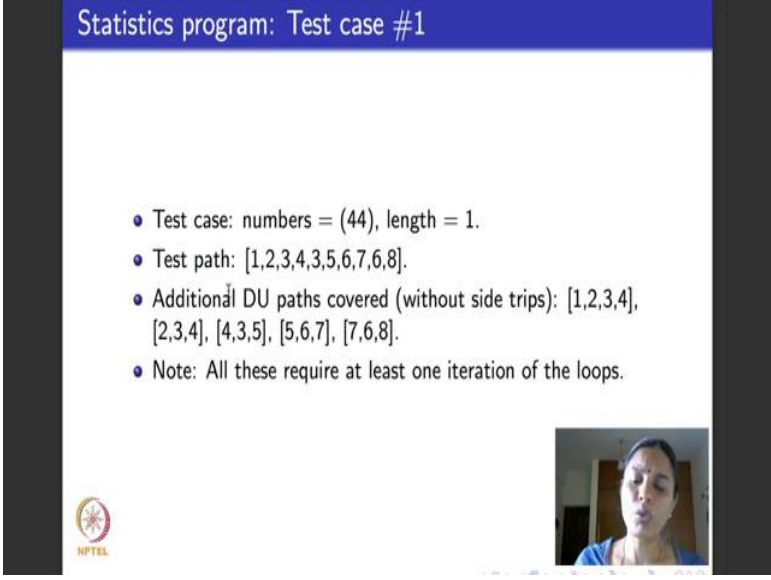
look at these paths then go back to the CFG and see if they really skip on loop. So, what are the paths? 1, 2, 3, 5, remember from 3 it goes to 5. So, similarly here from 1 to 3 it again goes to 5 and then does something from 3 it again goes to 5, and from 5 it goes to 6 and 8. So, we will go back now to the CFG and trace out these paths. So, 1, 2, 3, 5 it does skip the for loop, then 1, 2, 3, 5, 6 enters, but it took then take 7 went to 8. So, it skipped this second for loop. These four paths correspond to paths that skip the loop right the remaining paths 1, 2, 3, 4; 1, 2, 3, 4, 6, 7; 4, 3, 4; 7, 6, 7; 4, 3, 5 and 7, 6, 8 what do they do. They require at least one iteration of the loop.

So, we will go back to the CFG 1, 2, 3, 4, I have entered the loop I do 3, 4, I have iterated because once I enter 4 the only way to come out is to execute the loop once, similarly once I enter 6 the only way to come up to execute the loop once, and I do not go back to 4, I go to 3 instead. So, these path need one iteration of the loop, the remaining paths which talk about 4, 3, 4 and 7, 6, 7 talk about walking in the loop. I am again going back to the CFG 4, 3, 4. So, when I do 4, 3, 4 I have come back into the loop. So, I am taking the second iteration of the loop.

Similarly, when I do 7, 6, 7, I have come back to the loop. So, I am taking a second iteration of the loop right what did I do? I have started with defs and then uses and then I listed the def use pairs and then for each def use pairs we listed the du-paths. Several of these du-paths were repetitive. So, there were only 12 unique du-paths, it just so happens that they correspond to some of them correspond to paths that skip a loop, some of them correspond to paths that require at least one iteration of the loop, and the rest of them correspond to paths that require 2 iterations of the loop. So, only these du-paths that we need to cover.

This information that I told you actually tell you what those paths mean. They actually mean something like prime path coverage or loop coverage here; because if you see they nicely skip the loop require at least 1 iteration of the loop and require more than 1 iteration of the loop right. Now what we will do is we will go ahead and write test cases that execute these d u paths right.

(Refer Slide Time: 25:48)



Statistics program: Test case #1

- Test case: numbers = (44), length = 1.
- Test path: [1,2,3,4,3,5,6,7,6,8].
- Additional DU paths covered (without side trips): [1,2,3,4], [2,3,4], [4,3,5], [5,6,7], [7,6,8].
- Note: All these require at least one iteration of the loops.

NPTEL

Video inset showing a speaker.

So, what do the test case look like? I want to focus on writing test cases for each of these groups. So, I will write test cases for paths that require at least one iteration of a loop, I will write test cases for paths that require 2 iterations in the loop and then finally, write test cases for paths that skip a loop.

There is a reason why I want to put this at the last it will become very clear soon. So the first test case that I write this for paths that require at least one iteration of the loop, it so turns out that a very simple test case is enough to do this. The simple test case is the array is just this, it as it has just one number. So, its length is 1 and the single number that it has let us say it is 44 some positive number. So, is it clear? The array is just an array of length 1 containing a single number 44, and what is the test path that it takes? It happens to take this path 1, 2, 3, 4, 3, 5, 6, 7, 6, 8. Please remember that this is a test path a test path by definition has to always begin in an initial node and end in a final node.

So, we take remember this if I do 1, 2, 3, 4, I am entering the loop right if you remember the CFG, I come back to 3 finish one iteration of the loop I go to node 5 which comes out of the loop then I do 6, 7, 6, 8. So, let us go back and give this code an input of 44. So, the numbers array is just an array of length 1 which contains the value 44; length will be computed as 1, will enter the for loop some will be computed as 44. Median mean varsum will be initialized, varsum will be computed.

So, in the CFG it will go through 1, 2, 3 do this for loop once we are just sum is computed once, it is added 44 is added to 0 come out compute the various things, and then what it will do, it will do 1 iteration of this for loop varsum is computed and come out. So, it helps to have an array of length 1 right, because I just want to add that 1 number to sum, execute the loop once and come out right that explains why I gave an array of length 1 as a test case to this.

So, this is a test path that it traces and which are the du-paths that it covers? It covers in my list of 12, it covers these paths that needed one iteration of the loop. It covers them correctly and. in fact, the nice thing is that it covers them without any side trips. It covers them directly, you do not need any side trips to towards this right. So, now, I want to write a test case that will cover paths that will need two iterations of the loop, which means what? What do the loops correspond to the first loop corresponds to iterating over the length of the array and adding the numbers to sum. So, I need an array of length more than one any array of length more than one would do.

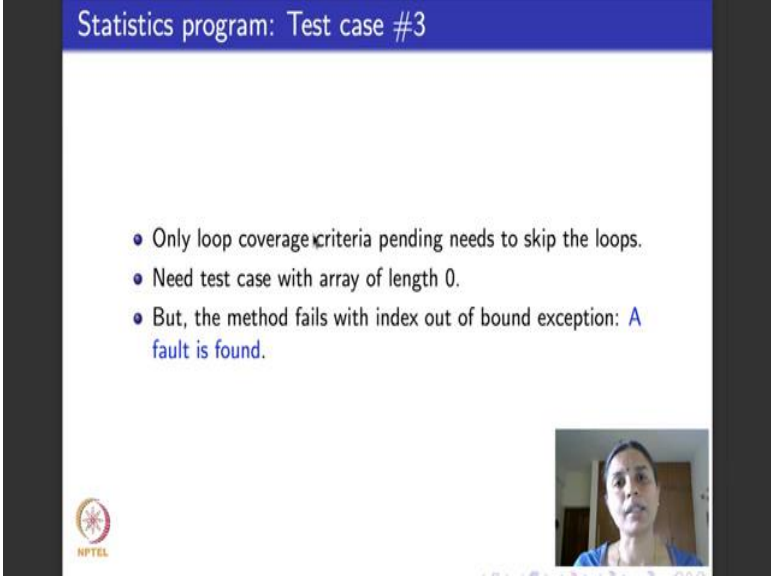
So, here is a sample array that I give as my test case, which has the numbers 2, 10 and 15 and it is length happens to be 3 right, because it has 3 numbers what it will do when it executes the code is, it will take this for loop 3 times and this for loop 3 times, because it has to go and add the 3 numbers that it finds across the array to the variable sum in this for loop, and in this for loop it has to compute varsum right.

So, what it will do is that you take the paths it will add the first number, it will add the second number, it will add the third number go through this for loop more than twice come again here, go through this for loop also more than twice because there are 3 numbers in my array right. So, the test paths that this one takes will look like this sorry will look like this. So, it does 1, 2, 3, 4, 3, 4, 3, 4 where the 3 numbers are added here in the 3 iterations of the loop first for loop, comes out of the first for loop then computes varsum by doing 3 iterations of the second for loop 6, 7; 6, 7; 6, 7 comes out of the second for loop at 8 it prints, right.

So, in this it covers the paths 4, 3, 4 and 7, 3, 7 which required at least 2 iterations of the loop. Now if I go back and take this segmentation of du-paths I have finished writing test cases for paths that require one iteration of the loop, finish writing test cases for paths that require 2 iterations of the loop, what is pending test case that first path that skips will

loop. So, if you see what will happen if we write a test case for path that is skip a loop what sort of a test case will that be; that will be an array of length 0 right.

(Refer Slide Time: 30:42)



Statistics program: Test case #3

- Only loop coverage criteria pending needs to skip the loops.
- Need test case with array of length 0.
- But, the method fails with index out of bound exception: A fault is found.

The slide features a blue header with the title 'Statistics program: Test case #3'. Below the header, there is a white area containing a bulleted list. The third bullet point is highlighted in blue. In the bottom right corner of the slide, there is a small video inset showing a person speaking. The NPTEL logo is visible in the bottom left corner of the slide.

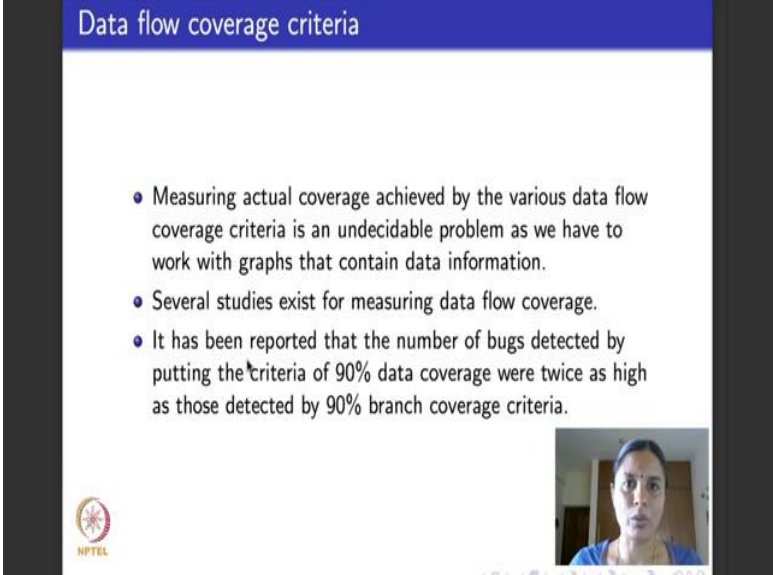
What will be a; what will happen if we give an array of length 0? Let us go back and look at the code. There will be a fault in the code now; probably the first fault that we are seeing through the span of this course. You will get to see little more faults which are always interesting to find in testing. So, if you see right here what will happen if I pass the parameter of length 0 right? I do this i pass a parameter of length 0 I come here initialize i to 0, i less than length, it will fail. So, an exception will be produced and the code is not meant to handle arrays of length 0, so the method will fail with index out of bound exception. So, I have found a fault.

So, what was the fault in the code? The fault in the code is very simple. The code did not handle the corner case of what will happen if an array of length 0 is given as input to the code. So, the code was computing all the statistical details correctly, it just did not handle one corner case and that is the fault that we have found by applying data flow coverage criteria to this code. So, I hope you walking through this example in detail would have helped you to learn how to apply data flow coverage criteria step by step, and see if the fault, if a fault if it is there in the code can be found.

It just so happens that for this example there was a fault and the applying data flow coverage criteria helped us to find the fault. We are more or less done with looking at

dataflow coverage criteria. I would like to recap some of the points that I told you last week, one is sometimes we are interested in asking how much have we covered in the code when I apply any of the three data flow coverage criteria.

(Refer Slide Time: 32:22)



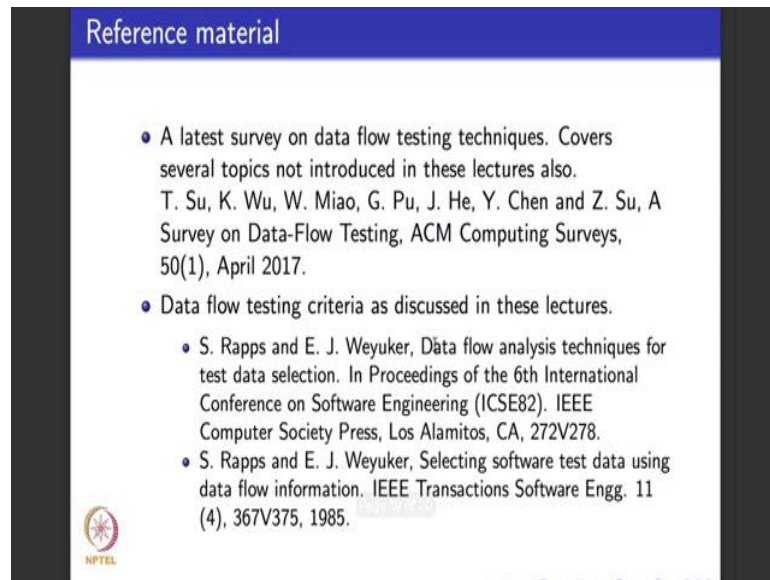
Data flow coverage criteria

- Measuring actual coverage achieved by the various data flow coverage criteria is an undecidable problem as we have to work with graphs that contain data information.
- Several studies exist for measuring data flow coverage.
- It has been reported that the number of bugs detected by putting the criteria of 90% data coverage were twice as high as those detected by 90% branch coverage criteria.

NPTEL

It so turns out that that is a tough problem to solve usually undecidable, and there are several studies that exist for measuring data flow coverage criteria. We will not be able to look at such measurements in detail, but empirically it is known that suppose I take one of the data flow coverage criteria and I insist that 90 percent of the data flow coverage criteria needs to be done right. It so happens that the coverage that it achieves is twice as high as those detected by insisting that I do 90 percent of branch coverage criteria. So, a data flow coverage criterion is indeed more effective and these are purely empirical studies.

(Refer Slide Time: 33:19)



Reference material

- A latest survey on data flow testing techniques. Covers several topics not introduced in these lectures also.
T. Su, K. Wu, W. Miao, G. Pu, J. He, Y. Chen and Z. Su, A Survey on Data-Flow Testing, ACM Computing Surveys, 50(1), April 2017.
- Data flow testing criteria as discussed in these lectures.
 - S. Rapps and E. J. Weyuker, Data flow analysis techniques for test data selection. In Proceedings of the 6th International Conference on Software Engineering (ICSE82). IEEE Computer Society Press, Los Alamitos, CA, 272V278.
 - S. Rapps and E. J. Weyuker, Selecting software test data using data flow information. IEEE Transactions Software Engg. 11 (4), 367V375, 1985.

NPTEL

You can look at these papers which I talked about at the end of the previous lectures also, to look at more details about data flow coverage criteria. These two are very seminal, old papers that introduced most of the data flow coverage criteria that we looked at throughout this course, and there is a latest survey on entire data flow testing techniques that is available.

So, what we will do next time is we move on to design. How to model design as graphs and how to look at data flow coverage criteria right. Just to end I took the statistical example from this text book that we have used throughout this course. So, when I see you for the next lecture we will move on from code to design and see how to apply graph coverage criteria for design.

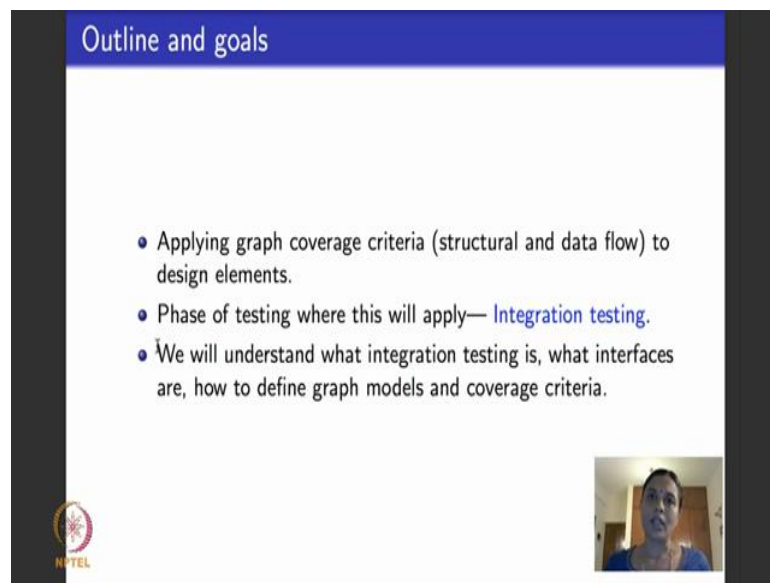
Thank you.

Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 16
Software Design and Integration Testing

Hello again, this is the second lecture of the fourth week. What I wanted to do today was to continue with graph based coverage criteria, we saw how to test code based on a graph coverage criteria. The basic idea was to take the control flow graph model it as a graph and then consider the various structural coverage criteria to be able to do graph coverage and then we augmented the CFG with definitions and uses, looked at du-pairs, du-paths and tested it for data flow criteria.

(Refer Slide Time: 00:47)



The slide has a blue header with the text "Outline and goals". Below the header, there are three bullet points:

- Applying graph coverage criteria (structural and data flow) to design elements.
- Phase of testing where this will apply— **Integration testing**.
- We will understand what integration testing is, what interfaces are, how to define graph models and coverage criteria.

In the bottom right corner, there is a small video inset showing a woman speaking. In the bottom left corner, there is a logo for NPTEL.

And what did we achieve by doing this? It was basically white box testing of code, we could cover the code see what statements were executed, how many times loops were executed, were loops skipped and so on, and through examples we saw that it was indeed useful to find the errors in codes.

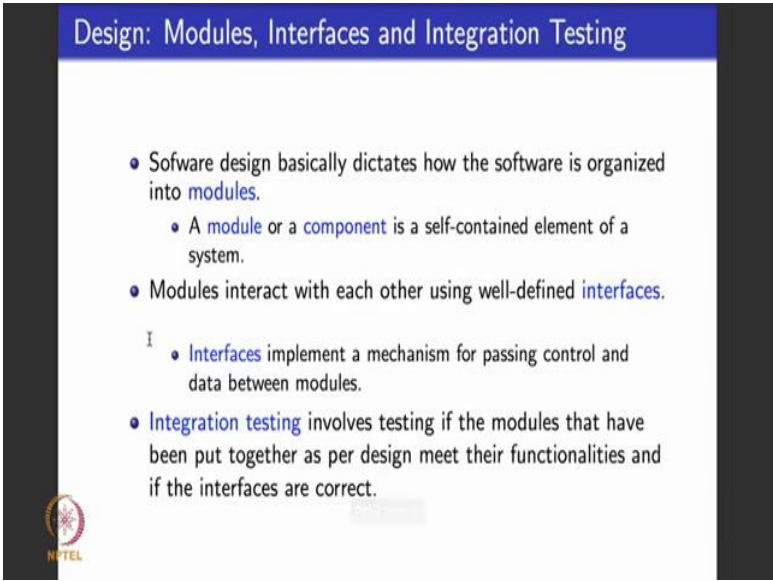
Now, I want to continue to use graph coverage criteria, but instead of considering code I want to be able to work with design, graphs for design models. When it comes to design the phase of testing that we are looking at is what is called integration testing. Integration testing basically assumes that design puts the software into various modules it puts

together these modules and test set. So, what I want to spend some time on with this lecture is to help you understand the traditional or the classical view of integration test.

What is integration testing? How has it been defined in the popular textbooks available on software testing, how do graphs corresponding to integration testing look like? So, today we will look at the classical view of integration testing any old book on software testing that is existed for many decades will use this view of integration testing. In the next lecture what I will tell you is I will tell you graph models for design and how to apply structural and dataflow coverage criteria on those graph models. We have to define slightly new structural and dataflow coverage criteria to be able to cater to graph models for design we will do that in the next lecture.

This lecture we will just spend on understanding integration testing as it is always been classically presented in several text books and software testing.

(Refer Slide Time: 02:30)



The slide has a blue header with the title "Design: Modules, Interfaces and Integration Testing". The main content is on a white background with a black border. It contains a bulleted list of definitions. In the bottom left corner, there is a small circular logo with a star and the text "NPTEL" below it.

- Software design basically dictates how the software is organized into **modules**.
 - A **module** or a **component** is a self-contained element of a system.
- Modules interact with each other using well-defined **interfaces**.
 - I • **Interfaces** implement a mechanism for passing control and data between modules.
- **Integration testing** involves testing if the modules that have been put together as per design meet their functionalities and if the interfaces are correct.

So, when I say software design, what is it? Software design for a large piece of software is basically tells you details about how its large piece of software is split into modules since the software runs into several thousands of lines, of code or even millions of lines of code. It is all not going to be one flat piece of code right; it is going to be split into several components each component is going to be designed and implemented by maybe by separate teams, separate individuals and then they are all going to be put together modular way to be able to constitute the whole piece of software.

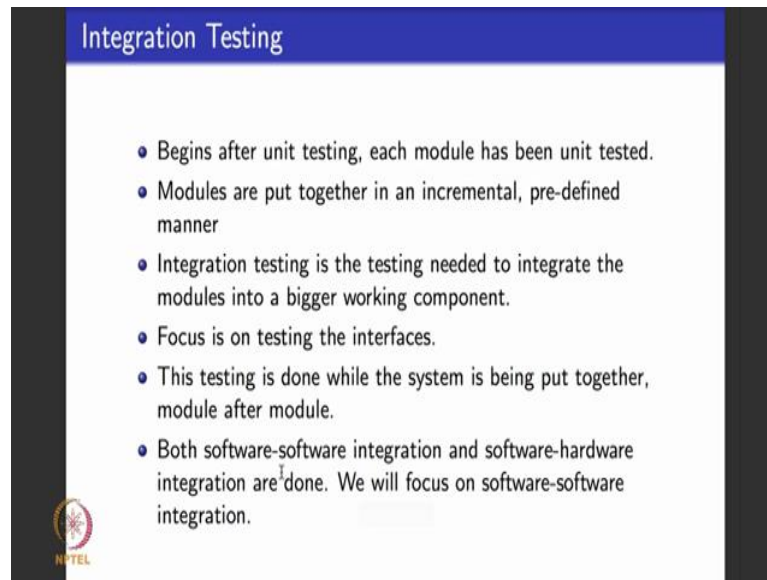
Typically for software, popular design languages are UML unified modeling language, SysML, system modeling language and some proprietary embedded software are modeled using proprietary design languages or even languages like Simulink, State flow etcetera. In this lecture we really will not consider each language for design and look at models for these languages. Instead I will assume that design is just a way that tells you how the software is split across modules and then we will see how these modules are put together and tested. So, what is a module? A module or a component, these terms will be used interchangeably, it is a self contained element of a piece of a software or a system.

What do I mean by self contained? By self contained, I mean that it takes inputs executes, and produces outputs. For its execution it is not dependent on another piece of software running or calling another piece of software or availability of data from another piece of software right. So, it executes as a standalone entity, stops its execution produces outputs and stops. Simplest view of a module would be a procedure in C or a method in Java right, individual method. Modules interact with each other using what are called interfaces. We will see what are the various kinds of interfaces, but interfaces always have to be well defined.

What is an interface? An interface you can think of it as implementing a mechanism for passing control and data between modules. So, one module can call another module or procedure or a function can call another procedure or function in which case it passes control to the other called procedure or function, and when it calls it can call it with certain data right it says you, please return this data to me or run it with this input and return the following input. So, it also exchanges data between the called procedure and the callee procedure, right.

So, interfaces are basically mechanisms that facilitate this call and return, control of call and return of data. Integration testing is basically a phase of testing that deals and tests if all the modules have been put together properly as per design and whether all the modules put together meet the functionality that they are supposed to meet together.

(Refer Slide Time: 05:36)



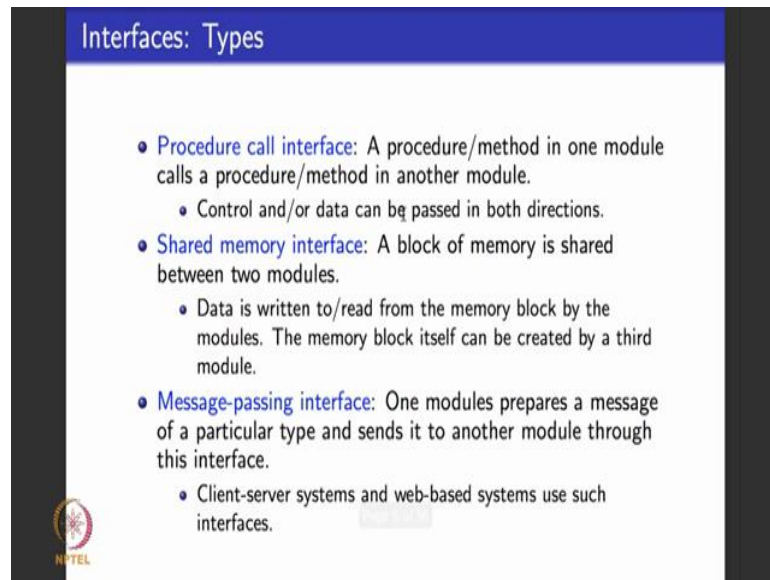
The slide is titled "Integration Testing" in a blue header. It contains a bulleted list of six points. In the bottom left corner, there is a circular logo with a red and yellow design, and the text "NPTEL" below it.

- Begins after unit testing, each module has been unit tested.
- Modules are put together in an incremental, pre-defined manner
- Integration testing is the testing needed to integrate the modules into a bigger working component.
- Focus is on testing the interfaces.
- This testing is done while the system is being put together, module after module.
- Both software-software integration and software-hardware integration are done. We will focus on software-software integration.

When is integration testing typically done? If you remember the phases of testing and the levels of testing that we saw right in the first week, integration testing immediately follows unit testing. When I mean integration testing here, I mean integration testing of software modules. There is also what is called software hardware integration testing which involves putting the piece of software in the hardware and then testing. That is not the focus for today's module. We mean software-software integration, putting together individual code components and testing the software right.

Modules can be put together in several different ways, but whatever; however, they are put together the way of putting them together is pre-determined and its incremental. It is not like I suppose I have 15 modules, we develop all the 15 modules and one fine they just sit and put together right, we do not do that. We have a well defined incremental way of putting together all the modules. Integration testing basically, what it does is that it test if the modules that have been put together are working fine. While testing for this, it focuses on testing the interfaces - how are the calls and returns happening? The focus is on that, designing test cases to see if there are any kind of errors on these interfaces. So, we spend some time understanding what are the various types of interfaces and how we will be dealing with them

(Refer Slide Time: 06:55)



The slide is titled "Interfaces: Types" in a blue header. It contains a bulleted list of three interface types. The first is "Procedure call interface", the second is "Shared memory interface", and the third is "Message-passing interface". Each has a sub-bullet describing its characteristics. An NPTEL logo is in the bottom left corner.

- **Procedure call interface:** A procedure/method in one module calls a procedure/method in another module.
 - Control and/or data can be passed in both directions.
- **Shared memory interface:** A block of memory is shared between two modules.
 - Data is written to/read from the memory block by the modules. The memory block itself can be created by a third module.
- **Message-passing interface:** One module prepares a message of a particular type and sends it to another module through this interface.
 - Client-server systems and web-based systems use such interfaces.

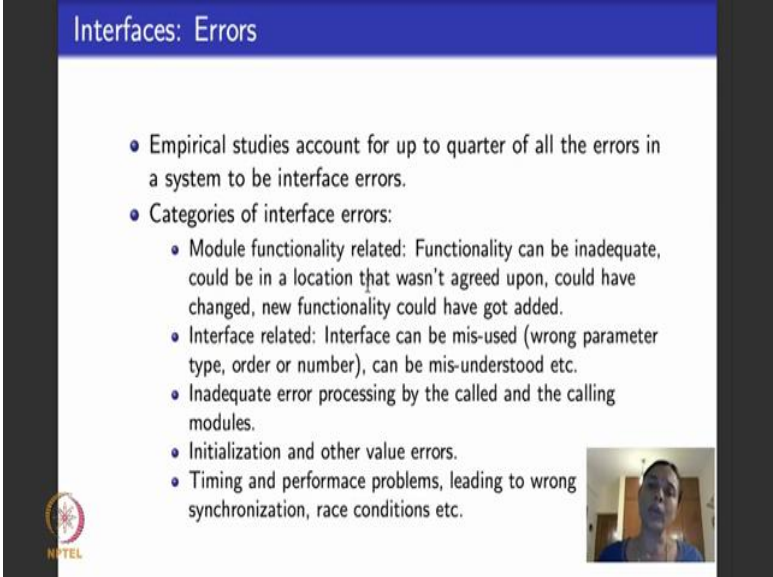
The most common interface is what is called procedure call interface. So, here we assume that a software procedure, a function or a method in one module calls another procedure or a method in another module right. This call could be called parameter passing, call by value, right several different ways that you would study in any standard C programming would apply to this kind of interfaces. Both control and data can be passed in both the directions. So, when it calls another module it transfers control to the other module and while it transfers control it also passes some data and when the call module returns back the control to the main module while returning back it could pass some data.

Second popular type of interface is what is called a shared memory interface. Here we assume that there is a block of memory that is available at some place. This block of memory could be within one of the modules that are sharing the interface or could be available or created by a third module in a different interface. How do these two modules communicate? They communicate by reading from that block of memory and writing onto that block of memory. So, this is a common shared global location and these modules communicate by reading from and writing to that value. Typically several systems software, like operating systems, embedded software, they all use these shared memory interfaces.

The third popular interface is what is called a message passing interface. Here what we do is that we assume that there are dedicated channels or buffers available and various components communicate by sending and receiving messages on these channels. Popular examples of systems that communicate with message passing interface are what are called client server systems. IoT is a very popular system these days that basically says that each individual IoT device is a client and then there are aggregator nodes, common nodes. So, it can be thought of as a kind of a client server system. Web apps, web applications that we all popularly use right, social networking applications like Facebook, Twitter and all other things, they also exchange a lot by using message passing interfaces. So, this gives you a broad idea of what the various interfaces could be.

But the testing methodologies that we would see today are independent of these kind of interfaces. This is just to educate you a bit about various interfaces that could actually exist. When we look at testing techniques for integration testing, for today's lecture we will abstract out and not worry about which is the kind of interface that is in play.

(Refer Slide Time: 09:47)



The slide is titled "Interfaces: Errors" in a blue header. It contains a bulleted list of error categories. In the bottom right corner, there is a small video inset showing a person speaking. The NPTEL logo is in the bottom left corner.

- Empirical studies account for up to quarter of all the errors in a system to be interface errors.
- Categories of interface errors:
 - Module functionality related: Functionality can be inadequate, could be in a location that wasn't agreed upon, could have changed, new functionality could have got added.
 - Interface related: Interface can be mis-used (wrong parameter type, order or number), can be mis-understood etc.
 - Inadequate error processing by the called and the calling modules.
 - Initialization and other value errors.
 - Timing and performance problems, leading to wrong synchronization, race conditions etc.

And why is it important to test for interfaces? It is important to test for interfaces because a lot of errors could come from interfaces. In fact, empirical studies in software engineering, empirical studies are studies based on collecting factual data right from various organizations which contribute to these data, we use lot of statistical tools to be

able to come to conclusions. Such empirical studies actually indicate that almost 25 percent, at most 25 percent of the total errors that come in software are basically related to interface errors, wrong calls wrong returns, whatever they are. What could be these kinds of errors?

So, here is a broad category of the various kinds of interface errors that can happen. It could be, the error could be related to module functionality in the sense that let us say one module is calling another module, it could assume that the module that is calling offers some kind of functionality for sure. But then it might be the case that the module that is being called has inadequate functionality, is not being able to offer the kind of functionality that it is assumed to offer, there could be errors because of that and there could be errors in where the module that is being called is located right. It could be in the wrong place, it could be in a place where it is not being able to call it easily right or the module that is calling could have been removed, could have been changed, a lot of things could have happened.

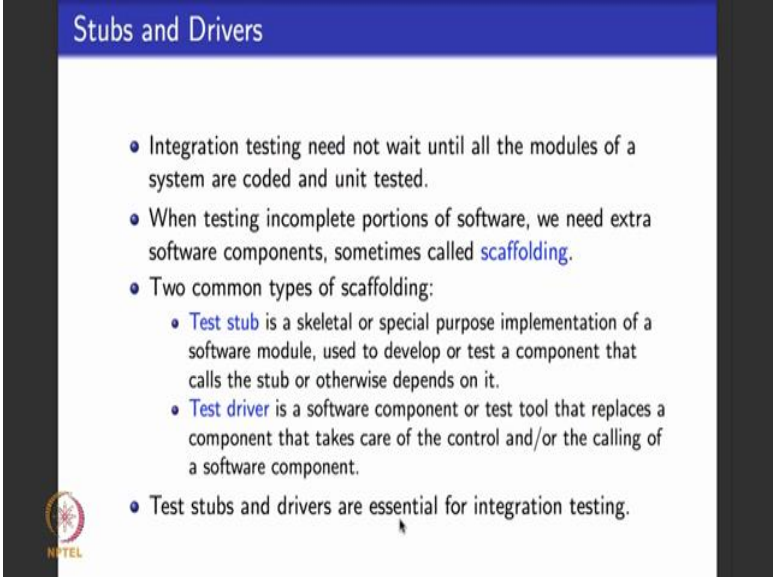
The next kind of error is the actual interface error itself in the sense that the interface can be misused. Suppose a procedure is calling another procedure let us say the first one procedure call interface, it could be the case that you are passing the wrong parameter type you are passing parameters in the wrong order, you have missed passing certain parameters, you are trying to pass more parameters, any kinds of errors could happen right.

The next is what is called inadequate error processing. So, here what we assume is that lets say suppose two modules are there one module is calling another module, you are also supposed to do some elementary debugging or error processing as a developer. Let us say the called module is error prone, the calling module is supposed to have a handler that is supposed to handle an erroneous message from the module that is returning the value. Maybe that was not correctly done right or it was assumed that certain kinds of errors would be there and those kinds of errors were not possible. So, there could be interface errors due to incorrect error handling.

The next is initialization and other value errors. You could pass wrong initial values, you could pass wrong data values, you could have missed initializing values anything can happen.

The last kind of popular interface errors are related to timing and performance. Sometimes you not only call a module you want to call module to respond to you fast enough, you wait for some time and if there is no response your timeout right and there could be errors because you have decided to timeout early or you waited for too long there could be errors related to these. But the basic summary is that interfaces are important, they can cause up to 25 percent of the errors that occur in typical software, so it is important to be able to test interfaces and integration testing focuses only on that.

(Refer Slide Time: 13:13)



The slide is titled "Stubs and Drivers" in a blue header. It contains a bulleted list of points about integration testing. The first point states that integration testing does not need to wait for all modules to be coded and unit tested. The second point explains that when testing incomplete portions of software, extra software components called "scaffolding" are needed. The third point lists two common types of scaffolding: "Test stub" (a skeletal or special purpose implementation of a software module) and "Test driver" (a software component or test tool that replaces a component that takes care of control and/or calling). The final point states that test stubs and drivers are essential for integration testing. An NPTEL logo is visible in the bottom left corner of the slide.

- Integration testing need not wait until all the modules of a system are coded and unit tested.
- When testing incomplete portions of software, we need extra software components, sometimes called **scaffolding**.
- Two common types of scaffolding:
 - **Test stub** is a skeletal or special purpose implementation of a software module, used to develop or test a component that calls the stub or otherwise depends on it.
 - **Test driver** is a software component or test tool that replaces a component that takes care of the control and/or the calling of a software component.
- Test stubs and drivers are essential for integration testing.

Before we move on and look at various approaches or methods of doing integration testing its useful to understand two other terminologies related to testing. As I told you right integration testing puts together the modules and tests them as and when they are ready. The typical belief is that integration testing need not wait until all the modules are ready because that will be too late right? Suppose you had the typical large systems that may have 100s of modules it does not make sense to wait through all the modules are ready. So, you test the system as and when the modules become available.

So, when you are testing the system as and when the modules become available you may not have the full test system that is test ready. So, what you have to do is a process in testing that is referred to as scaffolding. So, what happens in typical scaffolding is that whichever portion of the software is missing or is incomplete, you try to substitute for it

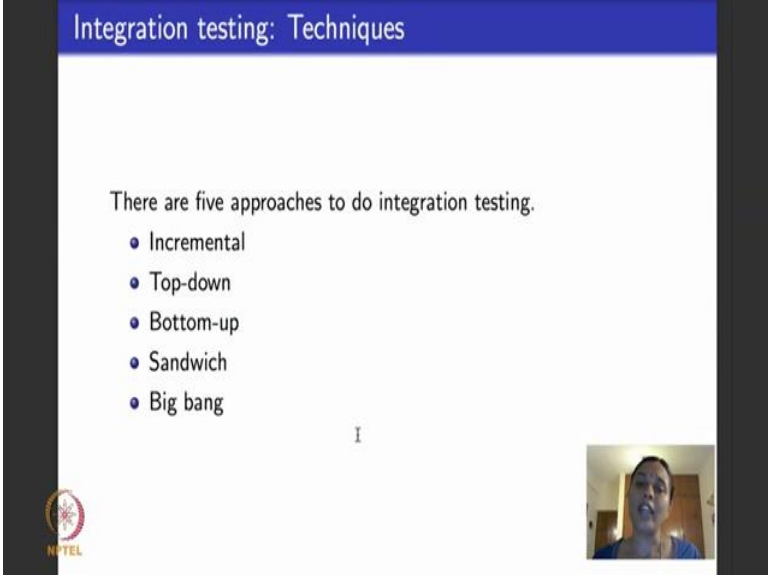
right. There are two kinds of popular scaffolding that is available--- one is what is called test stub and the other is what is called a test driver.

What is a test stub? The word stub says that suppose you had a module which you wanted a module and you did not really have it. So, you create a dummy module which behaves as if the module that you wanted was there, right? It takes some values it returns maybe some dummy values, but it does not represent the module. It is just a dummy entity that is being substituted for the actual module right. The dummy entity could need not represent the actual module, it can differ a lot from the actual module that is all right, but we still need a dummy entity to be able to go ahead and test. So, when I substitute an actual module with a dummy entity that behaves like the actual module in terms of interacting with the interfaces I have developed what is called it test stub.

The next kind of scaffolding is what is called a test driver. What is the test driver? That is the software component that replaces a component that is supposed to take control of the calling of the software component. Suppose you have three modules that have to be tested for integration testing, you want to put together and test. It so happens that three modules are being called by the parent module let us say one after the other, but the parent module itself may not be ready. So, then what you do is that you create a dummy parent module, it is called a test driver and make this dummy parent module just call these three modules right. So, when you do that then that kind of scaffolding that you develop is what is called a test driver. They are very very needed for integration testing it cannot integration testing cannot happen without stubs and drivers.

So, just to succinctly repeat what I said till now. What is a stub? A stub can be thought of as a module that is like a dummy, that represents the actual module. What is a test driver? The test driver is a dummy, but not for replacement of any module, it is a dummy module that calls the other modules that are in the lower level than this

(Refer Slide Time: 16:37)



Integration testing: Techniques

There are five approaches to do integration testing.

- Incremental
- Top-down
- Bottom-up
- Sandwich
- Big bang

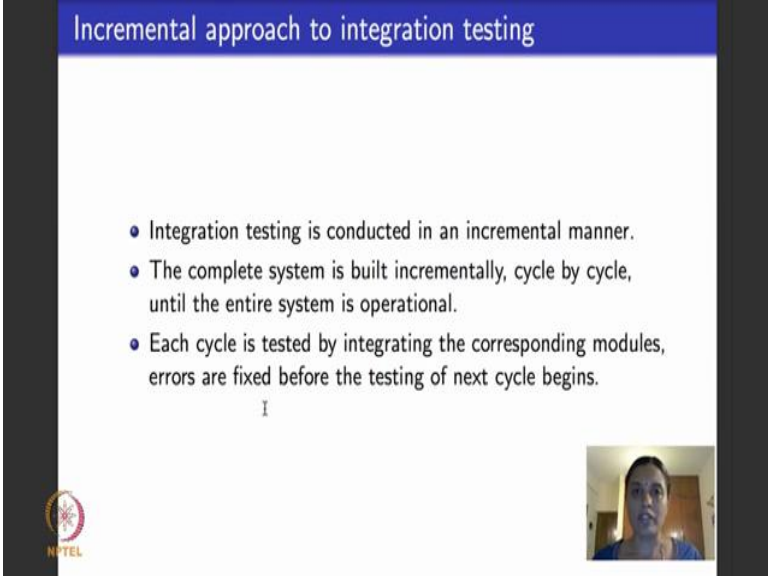
I

NPTEL

A small video inset in the bottom right corner shows a woman speaking.

So, there are five broad approaches to classical integration testing. Most of the books in software testing would refer to these approaches, the older books. The five broad approaches are incremental testing, top down integration testing, bottom up integration testing, sandwich testing and big bang testing.

(Refer Slide Time: 17:00)



Incremental approach to integration testing

- Integration testing is conducted in an incremental manner.
- The complete system is built incrementally, cycle by cycle, until the entire system is operational.
- Each cycle is tested by integrating the corresponding modules, errors are fixed before the testing of next cycle begins.

I

NPTEL

A small video inset in the bottom right corner shows a woman speaking.

So, we look at them one after the other. What is incremental testing approach to integration testing? So, as the word says, incremental means you do it in an incremental fashion, as and when you move, you keep integrating and testing.



So, the complete system is built incrementally phase by phase or cycle by cycle. As and when the modules of one cycle are ready I put them together, do integration testing. The next cycle is ready I put them together do integration testing and each cycle is tested for modules working in that cycle together, errors are fixed then and there and what is done later on is incremental testing, the cycle that has been put together first is not retested all over again.

(Refer Slide Time: 17:50)

Top-down approach to integration testing

- Works well for systems with hierarchical design.
- In hierarchical design, there is a first top-level module, which is decomposed into some second-level modules, some of which, are in turn, decomposed into third-level modules and so on.
- Terminal modules are those that are not decomposed and can occur at any level.
- Module hierarchy document is the reference document.

Three levels.
Seven modules.



Another popular approach to integration testing is what is called top down integration testing. So, here it assumes that the system design that breaks up the software into its various modules is hierarchical. So, it assumes the modules are organized in a hierarchy of levels.

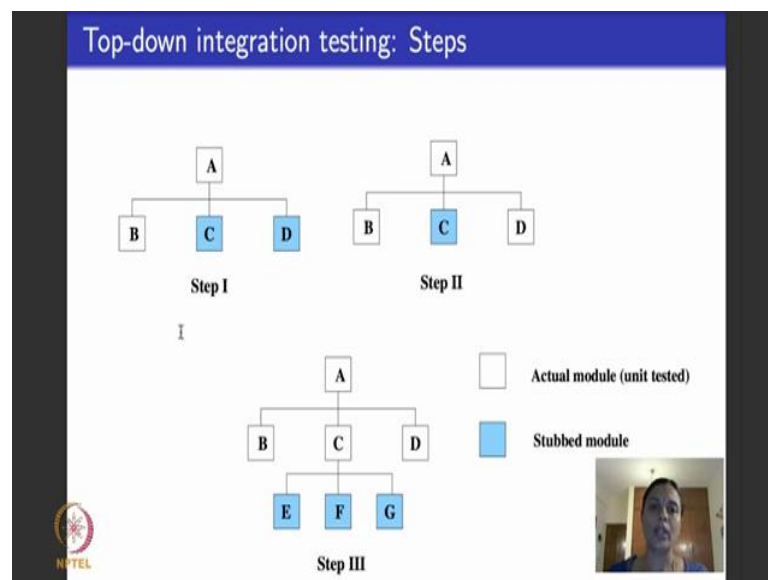
So, here is a small example. If you refer to this figure here, there are 7 modules in this figure: A, B, C, D, E, F, G and as you see the modules are arranged as if they were the vertices of a tree right. These vertices of a tree represent implicit hierarchy that is available in the module. So, here what it means is that there are three levels of hierarchy modules E, F and G are at the lowest level of the hierarchy, C calls these modules E, F and G, modules B, C and D are at the next high level of hierarchy and module A is at the topmost level of the hierarchy and that calls all the three modules B, C and D.

It so happens that in this case the figure looks fairly balanced and complete, at every level there are equal number of modules, but it may not be like that. What I meant is that

there are several modules organized into levels. At each level, the modules from the previous level call the modules in this level right. So, it has a tree or a directed acyclic graph structure to it. So, the terminal modules are the modules that occur in the leaves of a graph. In this case B, E, F, G and D are the terminal modules, A is a top level module, C is an intermediate module right. This is my reference document to begin the next phase which is top down integration testing and bottom up integration testing for that matter.

So, I have such a structure. What do I do when I do top down integration testing? So, there is a top and I start from the top which is the module A and I go down as I do integration testing. So, that is what is top down integration testing.

(Refer Slide Time: 20:05)



So, what I do? I will explain it in these figures. So, this is the total software design. There are three levels, 7 modules, I want to put them together one after the other, one after the other and test, and I want to put them together and test in a top down way. How do I go about doing it? I initially, I start with the topmost module which is the module A. Module A calls modules B, C and D right, it is not wise to put all B, C and D together and test A, I would want to test how A works with B, how A works with C and how A works with D independent of each other and I want to test how A works B, C and D together.

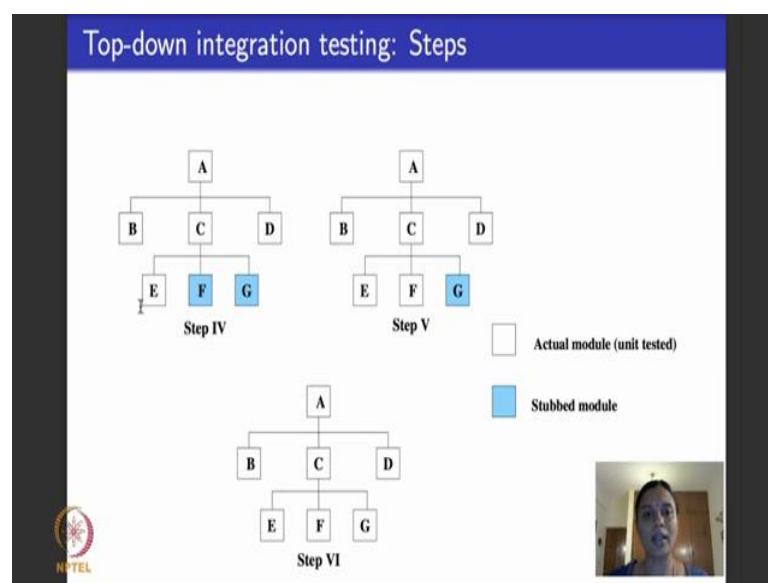
So, what I do first is that I will keep A, A is unit tested, ready. I keep B which is a unit tested, ready actual module, what I do is I create stubs for C and D, I create test stubs for

C and D. So, even if actual module C and A are ready I do not put them now, I just made them a stubs, dummies and why, because my focus is to test the interface between A and B alone. I test this part and now what I do assuming that have tested, all bugs if found fixed. Now I move on to testing the next interface I could test the interface between A and C or A and D. In this particular example, I am testing the interface between A and D, order does not matter, you could do A and C before you do A and D, there is no problem.

So, if I test interface between A and D, it is not wise to stub B now, because B is tested integrated with A, so I retain it as it is. I remove the stub for D, I replace D with the actual module right and then I test whether the interface between A and D as added to the interface between A and B which was already tested works fine. And the third step, what I do is I replace C with the actual model right. Now remember in our graph in this figure, C is a module that in turn calls E, F and G right. So, when I replace C with an actual module it comes as it calls for E, F and G, but for now I still want to focus and test if the interface between A and C is working fine. So, I do not keep the module C, E, F and G actual, I stub them out or they may not be ready, I do not really worry about it I just stub it out now.

Now, at this phase what am I testing? I am testing the interface between A and C to be working fine after having tested the interface between A and B, and A and D right. I still have to go on because I have to remove these three stubs and carry on.

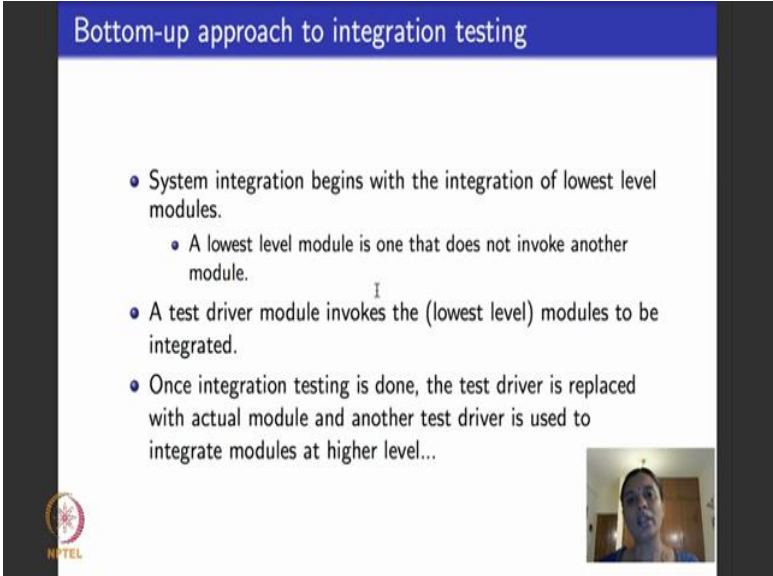
(Refer Slide Time: 23:10)



So, I can remove them in any order, here I have removed this stub for E, replaced it with the actual module for E, now I am testing this interface between C and E. And I move on I remove the stub for F, replace it with the actual module for F, test the interface between C and F and the last step I remove the stub for G, replace it with the actual module G and interface test the whole system. So, this is what is called a top down approach to integration testing.

I begin with the topmost module A, then I go and test the second level modules B, C and D. I can test them in any order, but typically I would want to do B and D because it testing C will involve moving to the third level. Between B and D I can test them in any order and then I move on to the next level which is testing C and here there are three more modules to be tested and because there is no further calls, I can test them in any order. So, I go from top all the way down and integrate the modules one after the other and test them. So, this is how top down integration works.

(Refer Slide Time: 24:17)



The slide is titled "Bottom-up approach to integration testing" in a blue header. It contains three bullet points: "System integration begins with the integration of lowest level modules." (with a sub-bullet: "A lowest level module is one that does not invoke another module."), "A test driver module invokes the (lowest level) modules to be integrated.", and "Once integration testing is done, the test driver is replaced with actual module and another test driver is used to integrate modules at higher level...". There is a small video inset in the bottom right corner showing a person speaking, and an NPTEL logo in the bottom left corner.

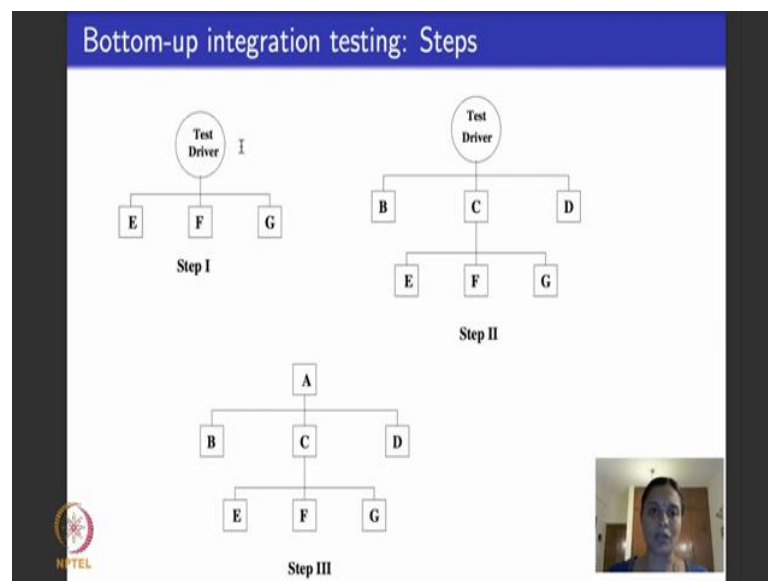
- System integration begins with the integration of lowest level modules.
 - A lowest level module is one that does not invoke another module.
- A test driver module invokes the (lowest level) modules to be integrated.
- Once integration testing is done, the test driver is replaced with actual module and another test driver is used to integrate modules at higher level...

The next approach to integration is what is called bottom up integration. So, here as the name says I start from the modules at the lowest level and keep moving up in the hierarchy till I reach the top most level. What is the lowest level module? It is a module that looks like this; E, F, G, B, D a lowest level models, they do not invoke any other modules right. Now to be able to put together the lowest level modules if you go back and see this figure E, F and G are the lowest level modules right? Suppose I want to be

able to test the interfaces for E, F and G how do I test them? E, F and G do not really call each other as for this figure right. Who calls E, F and G? It is actually C that calls E, F and G right.

But my goal is to go bottom up. If I use this test directly C, E, F, G and violating the approach of bottom up integration testing, I test the bottom most level with one level top which is C, I do not want to do that. So, what I now do is I create a dummy for C right. What is the dummy for C? I need a dummy that behaves like C in terms of calling modules E, F and G such a dummy is what is called a test driver. So, I create a test driver module that invokes the lowest level modules to be integrated. I test for that level, lowest and one level up and then this test driver module is actually replaced with the full module and then I go one level high and introduce the next test driver. So, that is what is illustrated in this figure.

(Refer Slide Time: 26:01)

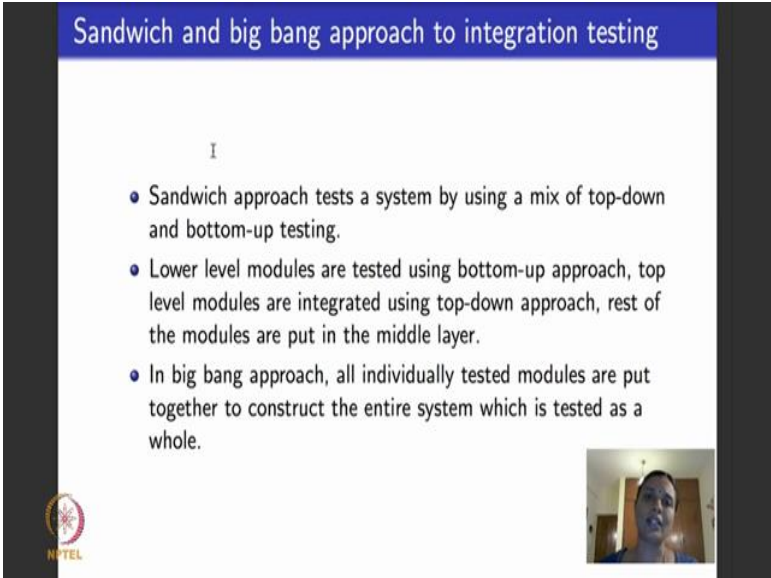


My goal is to do bottom up I start from the lowest level module. So, I start from E, F and G, I do not want to put C as it is. So, I write a test driver for C that calls modules E, F and G. And in the next step I replace C, the test driver with actual full module C. Now my idea is to integrate the test B, C and D. I still do not want to include A because A comes at next level above in the hierarchy. So, I now write a test driver for A and then test the integration of modules B, C and D at the second level. Finally, I replace the test

driver for a with the actual module and integrate test the whole system. So, this is how top down and bottom up testing works.

For top down, to recap, I need stubs to test one interface at a time. Stubs are dummy modules that replays other modules, just returns some dummy values or fixed values that are consistent with the module values that it would return and I do top down, start from the topmost modules and keep going down my hierarchy. In bottom up, I start from the lowest level modules and keep going up then the hierarchy, but to integrate test the lowest level modules I need to be able to write test drivers which are dummies, which will for the name sake, call the modules at each level that have to be integration tested. These two are the most popular integration testing techniques.

(Refer Slide Time: 27:41)



Sandwich and big bang approach to integration testing

I

- Sandwich approach tests a system by using a mix of top-down and bottom-up testing.
- Lower level modules are tested using bottom-up approach, top level modules are integrated using top-down approach, rest of the modules are put in the middle layer.
- In big bang approach, all individually tested modules are put together to construct the entire system which is tested as a whole.

NPTEL

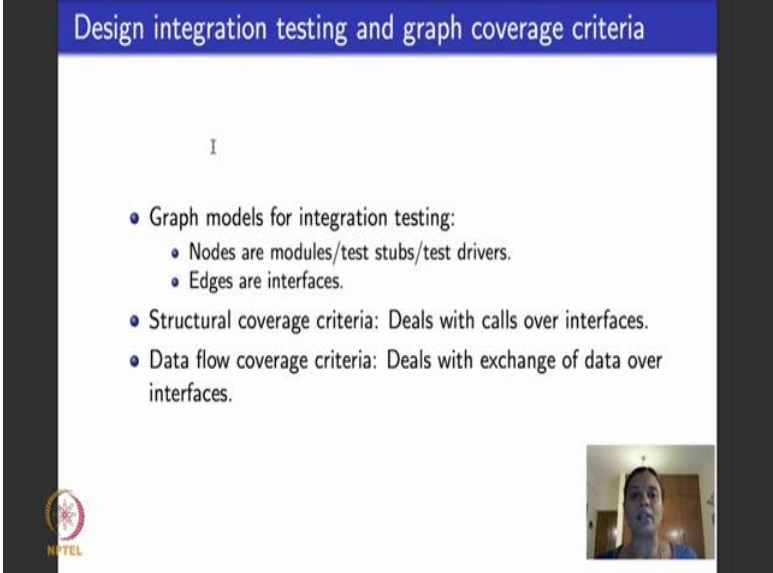
Video inset showing a person speaking.

There are other popular techniques called sandwich and big bang. So, what does sandwich do? As the name suggests its somewhere between top down and bottom up. Sometimes it uses top down, sometimes it uses bottom up, sometimes it uses a mix of them. It could be the case of the lower level modules are tested using bottom up the higher level modules are tested using top down and then they are put together in a sandwiched way.

The next popular approach to integration testing is what is called big bang testing. All individually tested modules are put together in one shot and tested. I am personally not a very big fan of big bang testing because I feel that it is very difficult to isolate a fault

when an error happens in a fairly large piece of software. I would advocate that when there is a clear cut notion of hierarchy, you follow either top down or bottom up based on which are the modules that are readily available for you to integrate and test.

(Refer Slide Time: 28:39)



The slide is titled "Design integration testing and graph coverage criteria" in a blue header. Below the title, the letter "I" is centered. A bulleted list follows:

- Graph models for integration testing:
 - Nodes are modules/test stubs/test drivers.
 - Edges are interfaces.
- Structural coverage criteria: Deals with calls over interfaces.
- Data flow coverage criteria: Deals with exchange of data over interfaces.

In the bottom right corner, there is a small video inset showing a person speaking. The NPTEL logo is visible in the bottom left corner of the slide area.

So, now what we will do in the next lecture is, we will go back to our favorite graph models. Our focus is now to look at graph models for design and see how they apply to integration testing as we learnt today. So, what do graph models for integration testing look like? If you see the ones that we saw here right these are graphs, all these are also graphs. So, what do the nodes in these graphs represent? They represent modules sometimes they could represent stubs or drivers. What do the edges represent? They represent interfaces.

As I told you, I really do not worry about which is the correct interface for now? When we go to later weeks look at object oriented software, web software at that time we worry about interfaces, but for now I consider them as just edges. Now we want to see how to apply structural coverage criteria which will deal with calls on the interfaces and how to apply data flow coverage criteria which will deal with parameter passing and return.

So, in the next module we will look at graph models for integration testing and specifically consider coverage criteria for all these things.

Thank you.

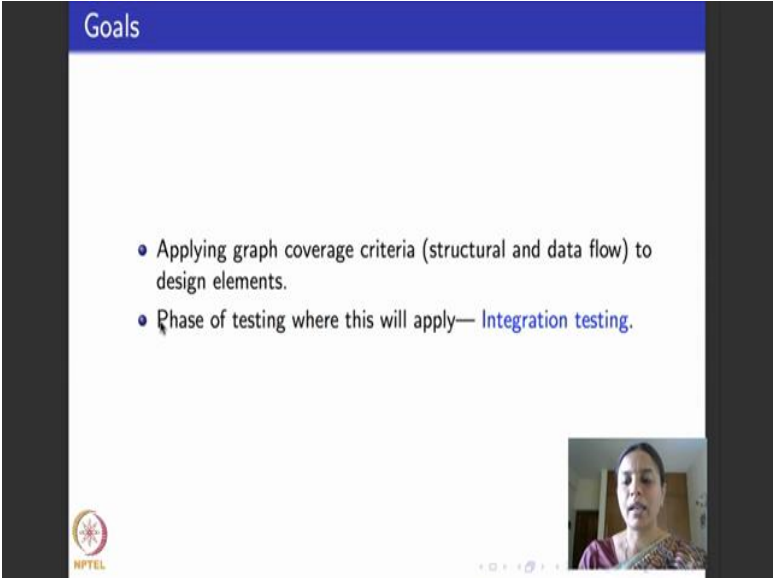
Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 17
Design Integration Testing and Graph Coverage

Hello again. This is the third lecture of fourth week. We are still looking at graph coverage criteria. What we did in the last lecture is we finally, moved out of looking at code and testing of code using graphs.

We started with design in the last lecture; I gave you the basics of what is the kind of testing which is done with design. The kind of testing that is done with design is called integration testing. We saw the classical view of integration testing, the various approaches to integration testing, what is tested and what is not. What we will do today is we will see how graphs can be used to do the kind of integration testing that we saw earlier. Specifically, we will see what are the coverage criteria that we have already seen on graphs that can be used for integration testing and what new coverage criteria could be defined over graphs when we look at integration testing.

(Refer Slide Time: 01:04)



Goals

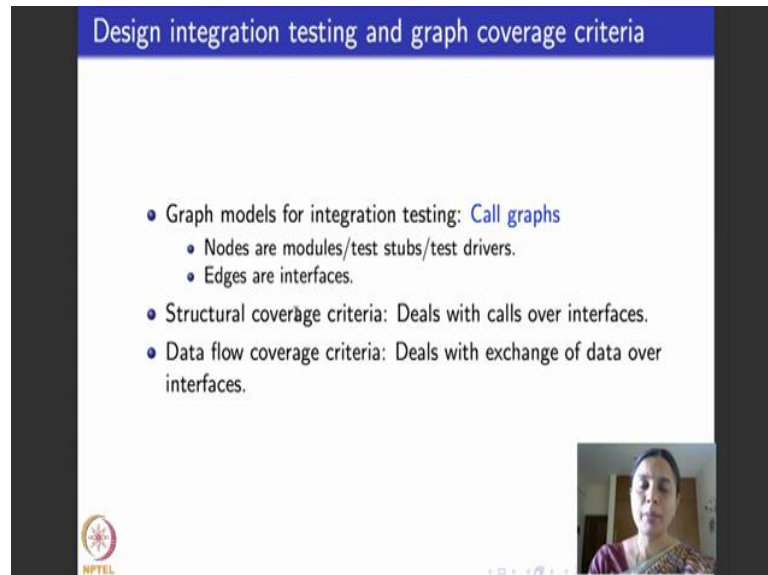
- Applying graph coverage criteria (structural and data flow) to design elements.
- Phase of testing where this will apply— **Integration testing**.

NPTEL

So, what are the goals for today's lecture? We want to be able to apply the graph coverage criteria that we have already learnt which is structural coverage criteria and

data flow coverage criteria, to be able to do design testing that is to be able to do integration testing that involves software design.

(Refer Slide Time: 01:23)



Design integration testing and graph coverage criteria

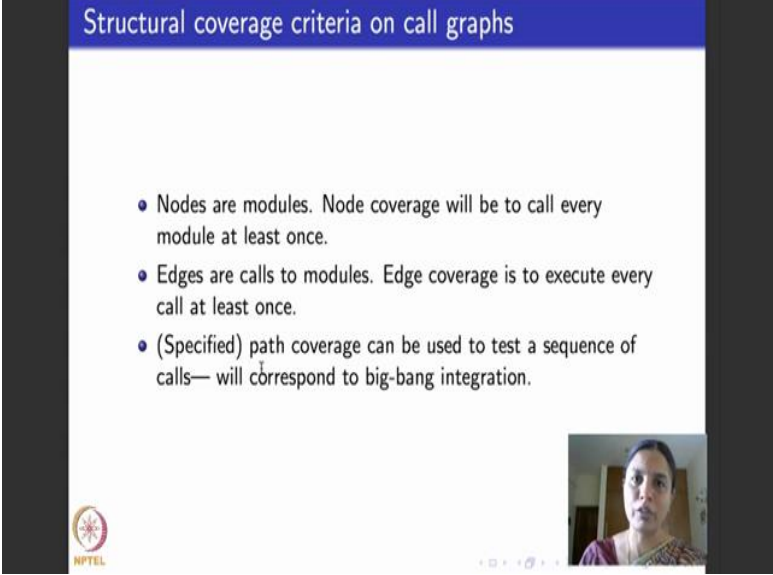
- Graph models for integration testing: **Call graphs**
 - Nodes are modules/test stubs/test drivers.
 - Edges are interfaces.
- Structural coverage criteria: Deals with calls over interfaces.
- Data flow coverage criteria: Deals with exchange of data over interfaces.

And while doing so, what will be the graph models that we will consider? The most popular graph models that we will consider are what are called call graphs as we saw last time. What is the call graph well there are several modules or components the software is broken up into these modules? And modules talk to each other through interfaces and when modules call each other or talk to each other through interfaces, the graph that models this is what is called call graph. The vertices or nodes of these call graphs are the various modules. Sometimes when I am doing bottom up or top down testing we saw that some modules could be replaced with test stubs or test drivers. So, the vertices of this call graph could be either the modules or the tests stubs or the test drivers.

The edges are basically the interfaces, the various calls. What do structural graph coverage criteria over such call graphs deals with it? So, happens that structural coverage criteria basically deal with calls over interfaces. You are basically checking if one module is calling every other module that it is supposed to. And the called module is it working fine right and so on. And then what you data flow coverage criteria over such call graphs deal with? They deal with how data is exchanged between the interfaces that connects these 2 modules.

Suppose it is a normal call interface where one module is calling another module. So, the data is passed as parameters the called module does its work and returns some data right. So, data flow coverage criteria deals with this passing and return of data across the interfaces.

(Refer Slide Time: 03:09)



The slide is titled "Structural coverage criteria on call graphs" in a blue header. It contains three bullet points:

- Nodes are modules. Node coverage will be to call every module at least once.
- Edges are calls to modules. Edge coverage is to execute every call at least once.
- (Specified) path coverage can be used to test a sequence of calls— will correspond to big-bang integration.

In the bottom right corner, there is a small video inset showing a woman speaking. The NPTEL logo is visible in the bottom left corner of the slide.

So, we will first look at structural coverage criteria then we will move on to data flow coverage criteria over call graphs. We are not looking at graphs that correspond to control flow right now. We are only looking at graphs that correspond to call graphs. Sometimes while modelling call graphs you might also have to model parts of the CFG, we will see through examples; how that is done.

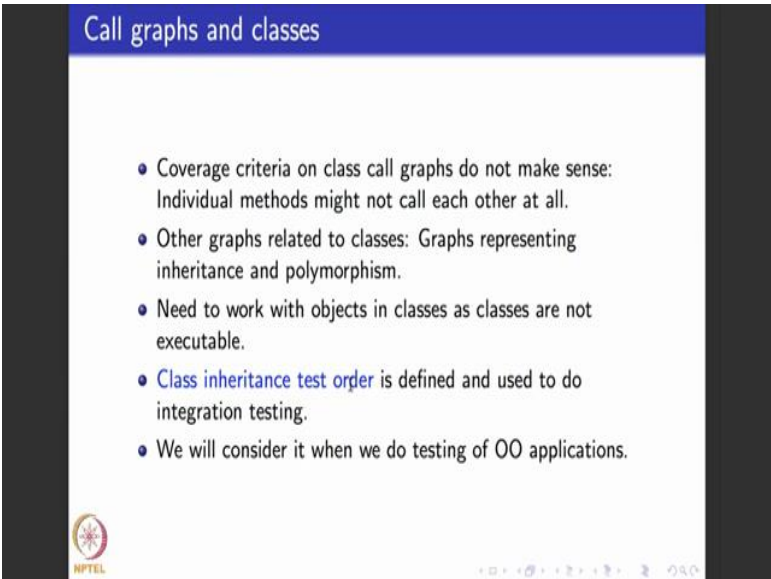
So, now what do structural coverage criteria over call graphs deal with? If you remember what are the various structural coverage criteria that we dealt with. We dealt with node coverage edge, edge pair coverage then complete path coverage which was infeasible then we looked at specified path coverage, prime path coverage. So, what is node coverage deal with here? What are nodes in our kinds of graphs when we do design integration testing? Nodes are basically software components or modules. So, node coverage would mean call every module that is supposed to be around right. So, suppose I have a procedure, that is a standalone procedure that is not being called by any other procedure or method, then it might as well not be there right. So, node coverage will

make sure that every component that is there in the modularized software is indeed useful and being called by some component.

What is edge coverage deal with, what are edges here ? As we saw edges for call graphs are interfaces. So, edge coverage deals with executing every call at least once. So, make sure that each call is executed at least one. So, it basically tests for coverage to make sure that there are no unnecessary calls unneeded, if there are calls that are never executed once then edge coverage will not be met and maybe it means that those modules can be removed, they are not needed at all, right.

So, edge coverage is a useful coverage criteria. Then it so turns out that other edge pair prime path coverage may not be very useful because if you remember the sole purpose of prime path coverage was to be able to handle loops in graphs. In call graphs we do not have loops and other things that we worry about. So, we do not consider prime paths coverage, but if you remember the various kinds of testing that we looked at--- top down, bottom up, big bang and so on in the last lecture. So, when I am doing big bang integration that is when I am putting together all the modules that I have developed so far and testing the integrated way of working together, then I can use what is called specified path coverage as structural coverage criteria to be able to do big bang coverage. So, this is one kind of use of structural coverage criteria.

(Refer Slide Time: 05:55)



Call graphs and classes

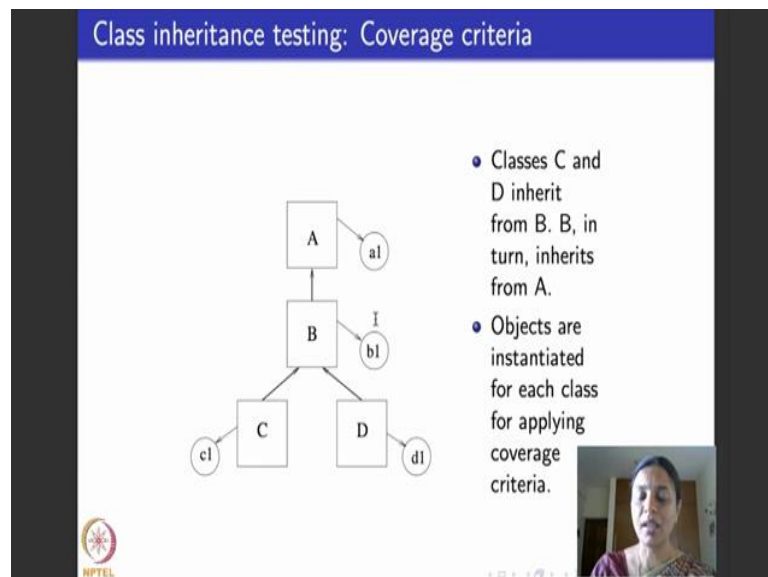
- Coverage criteria on class call graphs do not make sense: Individual methods might not call each other at all.
- Other graphs related to classes: Graphs representing inheritance and polymorphism.
- Need to work with objects in classes as classes are not executable.
- Class inheritance test order is defined and used to do integration testing.
- We will consider it when we do testing of OO applications.

NPTel

What other kinds of call graphs could be there in software. In object oriented software there are what are called class call graphs. And now we can say do we look at coverage criteria over class call graphs? But I would say that coverage criteria over class call graphs does not make sense at all. Why? Because individual methods residing within the class may not call each other at all. Classes are not executable objects. So, what could be other kinds of graphs that are related to classes? Typical other kinds of graphs that come related to classes are graphs that deals with class inheritance and iso and polymorphism and so on.

So, what we will do later in subsequent weeks after we look at all the various coverage criteria? We will spend one week where we will exclusively look at testing object oriented applications. In that week we will revisit this class call graphs and I will tell you lot in detail about how to test the large piece of object oriented software for functionalities like inheritance, polymorphism and so on. So, there is this theory called class inheritance test order that people define to be able to do integration testing over classes, but we will not look at it today. We will look at it after a few weeks then we dedicatedly look at object oriented testing concepts.

(Refer Slide Time: 07:22)

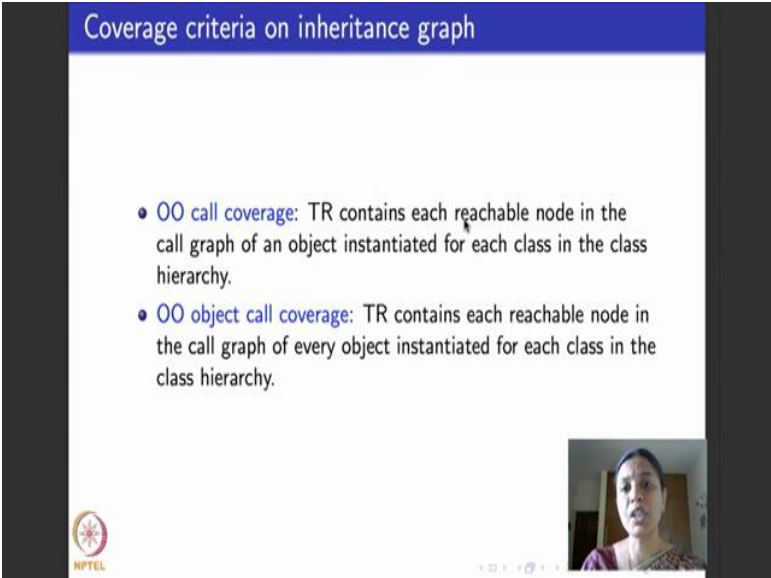


Right moving on when I do class inheritance testing in terms of coverage criteria, I could do the following. Consider this picture, if you look at this picture there are 4 classes here A, B, C and D. And read this connection arrow as inheritance. So, classes C and D

inherit from B, B in turn inherits from class A. As I told you classes are not executable. So, I really cannot do in terms of testing anything for this inheritance. Instead what I do is I consider objects that are instantiated for each of these classes before applying coverage criteria. So, in this picture I have depicted an object a1 that is instantiated for class capital A object b1 that is instantiated for class capital B object d1 for class D object c1 for class C.

So, this if this is the picture depicting an inheritance a hierarchy over class graph, how do I apply structural coverage criteria over such an inheritance hierarchy? So, it does not make sense to consider plane classes and apply inheritance hierarchy, I look at objects instantiated for each class and then apply inheritance hierarchy based testing.

(Refer Slide Time: 08:44)



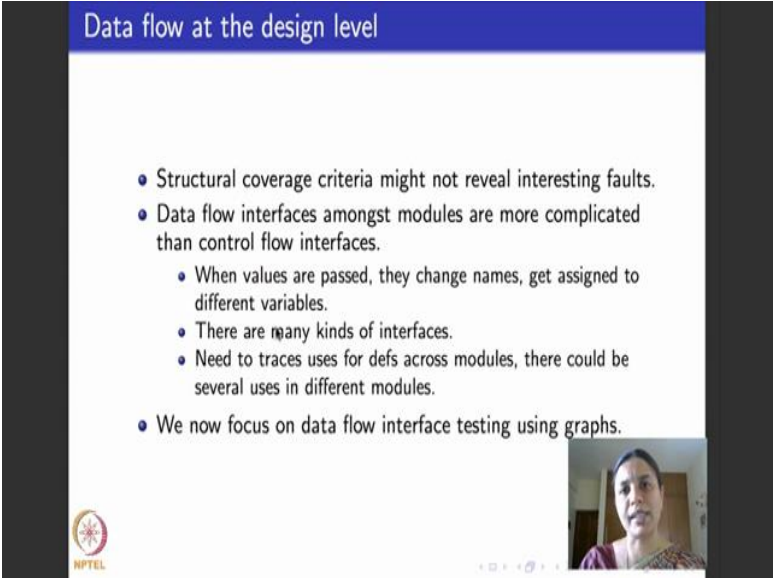
The slide is titled "Coverage criteria on inheritance graph" in a blue header. It contains two bullet points:

- **OO call coverage:** TR contains each reachable node in the call graph of an object instantiated for each class in the class hierarchy.
- **OO object call coverage:** TR contains each reachable node in the call graph of every object instantiated for each class in the class hierarchy.

In the bottom right corner, there is a small video inset showing a woman speaking. The NPTEL logo is visible in the bottom left corner of the slide area.

So, when I do that 2 kinds of coverage criteria makes sense the first kind of coverage criteria is what is called object oriented call coverage. So, we will see what it says, it says the test requirement or TR for OO call coverage contains each reachable node in the call graph of an object instantiated for each class in the class hierarchy.

(Refer Slide Time: 09:20)



Data flow at the design level

- Structural coverage criteria might not reveal interesting faults.
- Data flow interfaces amongst modules are more complicated than control flow interfaces.
 - When values are passed, they change names, get assigned to different variables.
 - There are many kinds of interfaces.
 - Need to trace uses for defs across modules, there could be several uses in different modules.
- We now focus on data flow interface testing using graphs.

NPTEL

So, I go back, I will look at this class hierarchy and then what do I say I say my test requirement is each reachable node in the call graph of an object instantiated for each class in the class hierarchy right. So, each reachable node should be covered, and the second one is object oriented of OO object call coverage that says the TR contains each reachable node in the call graph for every object instantiated for each class in the class hierarchy. So, every, so they here this picture depicts only one object instantiated per class, but that need not be the case there could be several of them. If that is the case then I do what is called OO object call coverage.

So, basically in summary what we are saying is that I look at class inheritance graph that looks like this do not consider classes plain because they are not executable. Instead you look at objects that are instantiated for each class. When I test the calls relate or when I test the calls related to objects in the call graph, one object per class, then it is called OO call coverage. When I test the calls related to objects instantiated in the class graph for each object in the class then it is called object call coverage. We will revisit as I told you all these concepts later in a week when we dedicatedly do object oriented testing applications.

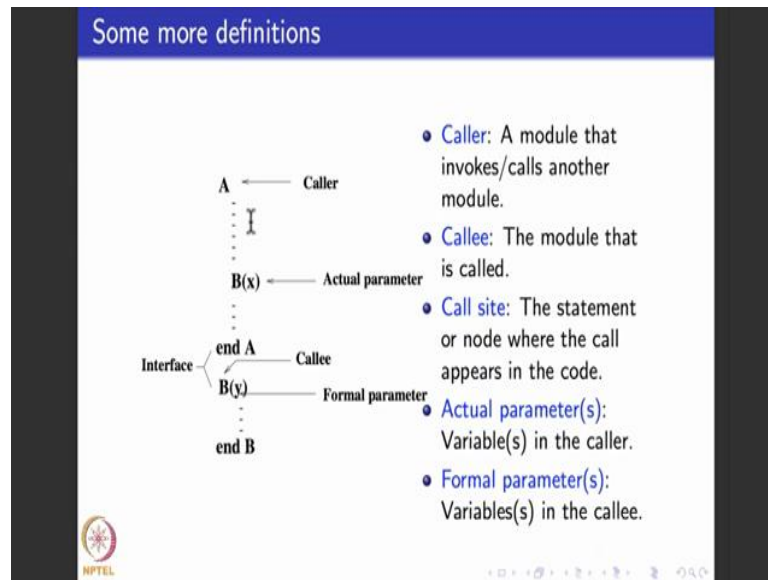
Now, they done with structural coverage criteria. I would like to move on to data flow coverage criteria over design. As I told you right data flow coverage criteria is very interesting when I look at inheritance testing because it actually deals with the

parameters that are passed and returned. Maybe that is a richer source of error instead of just checking for structural coverage criteria, we just check is a module called, is a module interface called and so on. It just checks whether it is called or not of course, that is also useful, but what is more useful is actually reasoning and testing for the values that are called and returned in each of these calls and returns. That is what data flow coverage criteria deals with.

So, data flow interfaces are more complicated than control flow interfaces, why? Because values are passed they; obviously, change names we will see it through examples I might pass a certain value in the variable name x, it gets passed to the callee as a variable let us say y. So, they change names they get assigned to different variables and as I, as we discussed in the last module there are several different kinds of interfaces, I could communicate by explicitly passing parameters and returning values. I could communicate by reading from and writing on to a global set of variables, I could communicate by sending and receiving messages across dedicated buffers. So, data flow testing has to be done across all these kinds of interfaces. And the other important thing is remember now definitions and uses or variables as we deal with in data flow criteria will have to run across modules. So, a particular variable could be defined in one module and used later in another module.

So, we will see through examples what each of these mean. So, the focus for the rest of today's lecture, this lecture would be to look at data flow coverage criteria, but specifically looking at data flow over call graphs as we use it first.

(Refer Slide Time: 12:26)



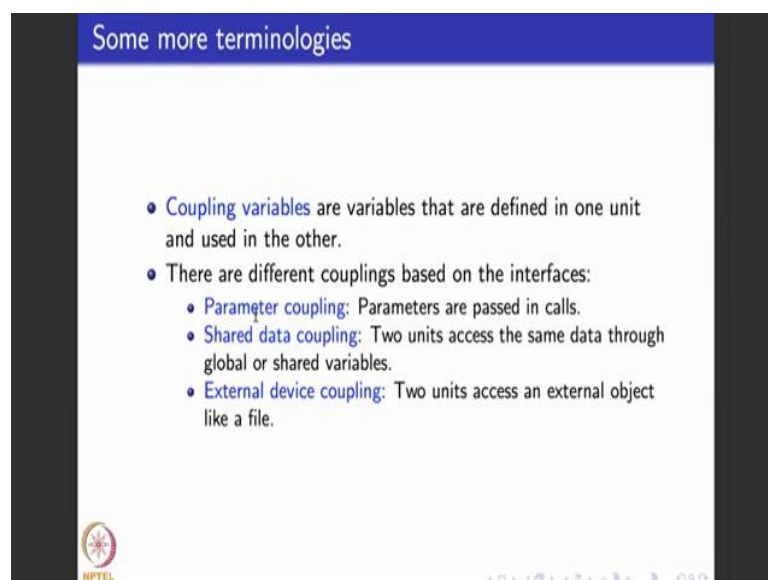
So, here is an example. Let us say if you look at the picture on the left hand side let us say there is a method A, a method or procedure A. A has some codes I am not really written what the code is read these dot, dot dots (...) that you see here is some code that corresponds to A as a part of the code that is written for A, A also calls another method or a procedure B with a parameter x.

When A executes and when this line of A comes into picture what happens, control gets transferred to the method or a procedure B, which is right here down there. And remember A calls B with actual parameter x and it is passed to B as y, which is the formal parameter. Strictly speaking x and y are the same value, but they take different variable names. I need to be able to remember the fact that x and y are the same values, but they take different variable names; x takes x in A and the same x is called y in B.

So, A runs, at some point in time A calls B with parameter x then x is passed to be as y then B executes. Read these dot, dot, dot (...) as some code that corresponds to B. I have not really written the code for A and B because the focus now is to test for the interfaces test for A calling B and B returning something. So, B runs something B finishes end B comes when B finishes the control is passed back to A and the rest of A executes and finishes. So, here are some terminologies that we will use for the rest of the couple of lectures.

So, A is called the caller or a module that calls another module in this case it is the module that it calls is B. B is called the callee module which is the module that is being called by A the call site is this statement the statement in a where the procedure call happens. Actual parameter is x, this x that is passed as a parameter to be, formal parameter is y the parameter that callee B takes from the caller A. Interface is the place where control is transferred from A to B and after B finishes running, control is transferred back to A. So, what are the terminologies that we looked at till now, this caller, this callee then there is call site which is the place where the call happens or the interface is connected. Then there are actual parameters then there are formal parameters.

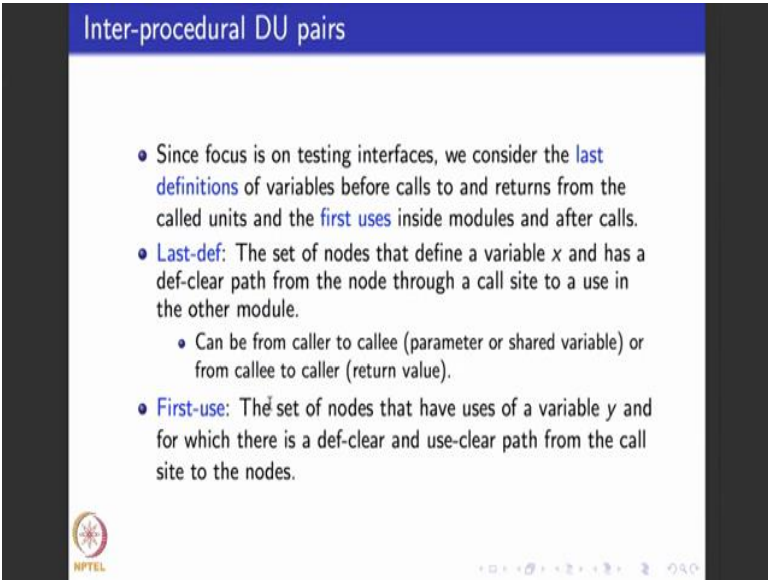
(Refer Slide Time: 15:02)



So, now if one more terminology before we move on if you go back to this figure, look at the actual parameter x. It is passed as x and B. What B take x as, B takes x as y. So, we say x and y are what are called coupling variables. So, coupling variables are variables that are defined in one unit and used in the other. And as we know based on the kind of interfaces that we are looking at there could be several different kinds coupling. There could be parameter coupling, example of parameter coupling is what we saw here. A calling B with x as a parameter and B is returning it back to a with some at something else. Coupling could be in terms of share data when modules communicate by reading from and writing to a set of global variables or shared variables or coupling could be in terms of an external device.

You could have a database that resides in a particular database server and then you say I read data from that database server and that could also deal with explicit coupling variables. But when we look at this particular module, I will assume that parameter coupling is one coupling that we are looking at and the abstract def use pairs that we will look at will deal mainly with parameter coupling and in other cases also it should go through at a same level of abstraction.

(Refer Slide Time: 16:23)



The slide is titled "Inter-procedural DU pairs" in a blue header. It contains three bullet points. The first bullet point states that since the focus is on testing interfaces, we consider the last definitions of variables before calls to and returns from the called units and the first uses inside modules and after calls. The second bullet point, "Last-def", defines it as the set of nodes that define a variable x and has a def-clear path from the node through a call site to a use in the other module, with a sub-bullet noting it can be from caller to callee (parameter or shared variable) or from callee to caller (return value). The third bullet point, "First-use", defines it as the set of nodes that have uses of a variable y and for which there is a def-clear and use-clear path from the call site to the nodes. The NPTEL logo is in the bottom left corner, and navigation icons are in the bottom right.

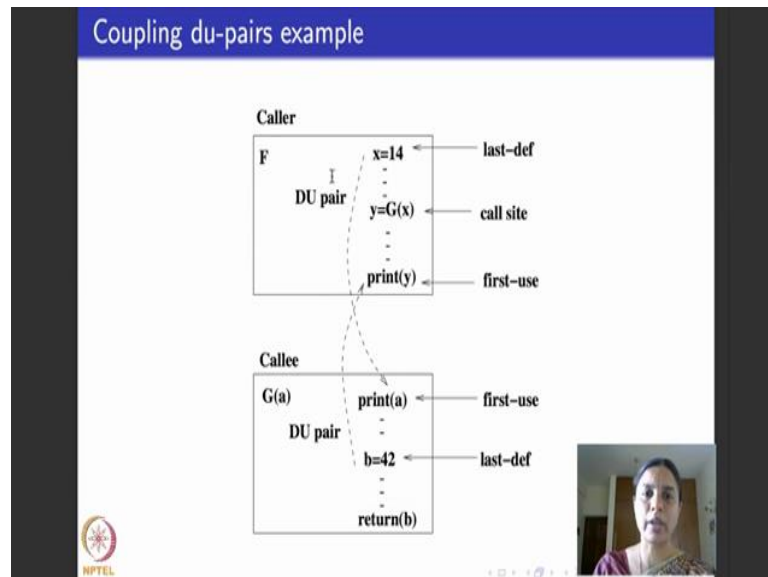
- Since focus is on testing interfaces, we consider the **last definitions** of variables before calls to and returns from the called units and the **first uses** inside modules and after calls.
- **Last-def:** The set of nodes that define a variable x and has a def-clear path from the node through a call site to a use in the other module.
 - Can be from caller to callee (parameter or shared variable) or from callee to caller (return value).
- **First-use:** The set of nodes that have uses of a variable y and for which there is a def-clear and use-clear path from the call site to the nodes.

Some more definitions before we move on and look at an example. Remember what is the focus, the focus is to be able to understand and test how data is passed across interfaces right. So, the focus is on interfaces. So, if I go here, if I consider data flow testing for A and data flow testing for B, there could be a whole set of variables that reside any A, whole set of other variables reside in B. Which I will do when I do normal data flow testing for source code, I will test it for that, but the goal for today is to be able to test how A calls B and how return B returns A.

So, what I do is I focus on the parameters that are passed and returned and the values that are defined and used around the parameters that are passed and returned. So, that gives us the definition of what are, what is called the last definition of variables which occur just before calls to the another module, and what is called first uses of variables which occur soon after the module is called, soon after the module is called inside the called module.

So, what is last def? Last definition is a set of nodes in the control flow graph of the caller module the define variable x and it has a def clear path from the node through the call site to a use in another module.

(Refer Slide Time: 17:53)



So, to better understand this let us look at an example. So, here is a caller, method or procedure F calls another method or procedure G. So, G is the callee, let us say F has some code that goes like this again, please I have read these dot dot dots (...) as some code that is there, but I am interested I have put the statements corresponding to only fragments of the code that I want to focus on. So, at some point in it is working F has the statement, where it says x is 14. And after that it is def clear for x, x is not defined in these places where I am pointing to, in these dot dots after x is equal to 14. And then a F calls G of x with this value of x and then control gets passed to G.

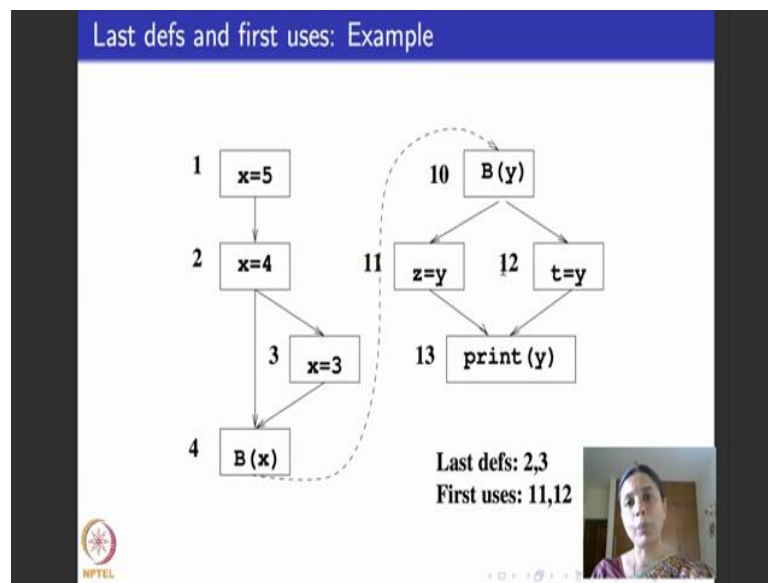
What is G do? G takes x as a, it runs as G(a). So, this x actual parameter is passed as a just a formal parameter to G the first thing that G does is to print the value that it obtains. So, the last def of x was here and the first use of x passed as a is in this printf statement. Is that clear? Please note that I do not consider the parameter passing itself as a use. So, you could argue that this could be thought of as the last use of x in F and this could be thought of as the first use of x as a, coupled with a in G. But typically when we do integration testing the point of parameter passing at the call site is not considered a use.

What we consider is after it is passed by the called method or procedure, inside the called method or procedure where is it first used.

So, this print a of x being passed as A is considered the first use of the variable x that is coupled with a and now G after prints a it executes a few more statements. At some point it defines b as something like this b is equal to 42 then it executes something this bit after the definition of b is def clear for b and then it returns b. When it returns b, what is b passed on as it is passed on here into the variable y because F call G of x and keeps it as y after some point F decides to print y right. So, the last def of b that was here gets first use as print y when b is coupled with y here again I do not consider this return statement as the first use of b. So, going back to the definition what is last def say? Last def says that a variable x has a def clear path from the node through the call site to a use in another module.

Now, this can be of two different kinds. It can be from a caller to a callee or it can be from a callee back to the caller. We saw examples of both these cases right. So, here if you see, this is from the caller to the callee, this last def x is equal to 14 is from the caller to the callee. This last def for b being 42 is from the callee back to the caller. So, two kinds of last defs can be there. Similarly, what is first use? First use, the set of nodes that have uses of variable y, for which there is a def clear and a use clear path from the call site to the nodes. So, there is a def clear and a use clear. So, if you see here I have mark the first uses also, because once this last def happens this is the place where it is first used and in between we assume that there are no further definitions are use. One thing to note is that this calling, and return at the call site does not constitute a part of the definition of last def or first use.

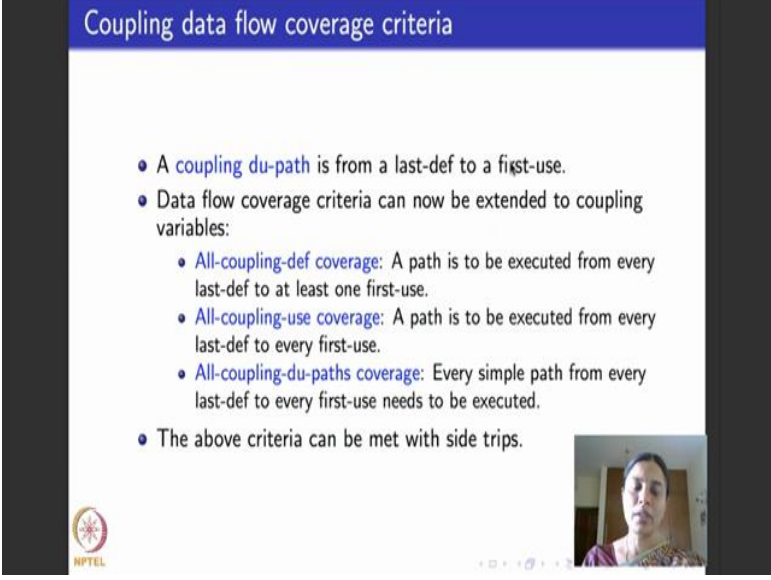
(Refer Slide Time: 22:11)



So, just a few more examples before we move on. So, here is another piece of code. I have depicted it differently this time. So, what we have shown is not as boxes like this containing the caller and the callee modules. What I have shown is I have shown it directly along with the control flow graph of the caller module and the callee module. So, the one that you see on the left hand side here; x is equal the which and vertices labelled with 1, 2, 3 and 4 this is part of the caller module. B is the caller module right and in this module is a caller module the callee module is B, the variables that are coupled are x and y B runs, maybe there is an F statement here one of these happens and then it prints y.

So, what are the last definitions? Last definitions for x could be either at node 4, I mean node 2 or node 3 based on what the decision of this statement is. And what are the first uses, not at the call site, please remember that, it is either at 11 or 12 based on what this condition is. So, is this clear?

(Refer Slide Time: 23:25)



The slide is titled "Coupling data flow coverage criteria" in a blue header. It contains a list of bullet points:

- A coupling du-path is from a last-def to a first-use.
- Data flow coverage criteria can now be extended to coupling variables:
 - All-coupling-def coverage: A path is to be executed from every last-def to at least one first-use.
 - All-coupling-use coverage: A path is to be executed from every last-def to every first-use.
 - All-coupling-du-paths coverage: Every simple path from every last-def to every first-use needs to be executed.
- The above criteria can be met with side trips.

In the bottom right corner, there is a small video inset showing a person speaking. The NPTEL logo is visible in the bottom left corner of the slide.

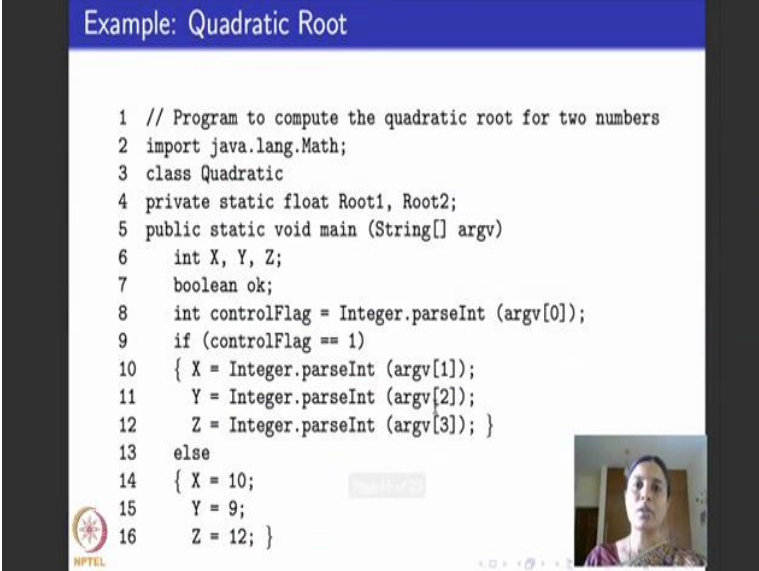
So, moving on, we had three data flow coverage criteria that we define for call graphs. What are the three data flow coverage criteria? If you remember there were all defs coverage, cover every definition to at least one use then it was all uses coverage, cover every definition to every use. And then the third was all du-paths coverage, from every definition take every kind of paths to every use.

So, I retain the same definitions instead I look at coupling variables. So, what is a coupling du-path, a coupling du-path is a path from not any F def to any use it is from the last def to a first use. So, I am in a situation where my code is fragmented into 2 modules. One module calls another module. So, in the caller module there is a place where last definition of a variable happens. And in the callee module there is a place where the first definition of a variable happens. Then this pair is what is called a coupling pair. Here x and y is a coupling pair. So, what I do is, I take coupling du-path as a path from last def to first use once I have coupling du-paths, I take the same definitions of data flow coverage criteria that we looked at earlier, and consider the same definition, but instead of considering it overall du-paths I consider it from last def to first use.

So, all coupling def coverage is a path to be executed from every last def to at least one first use. All coupling use coverage is a path to be executed from every last def to every first use. All coupling du-paths coverage says that execute every simple path from every last def to every first use. As always you do best effort touring. So, whenever I have to

meet this data flow coverage criteria, if I have to do a side trip or a detour I can do that to be able to make these test requirements satisfiable. Of course, the one requirement that I have about side trips, to repeat, is that side trips have to be definition clear right.

(Refer Slide Time: 25:37)



Example: Quadratic Root


```
1 // Program to compute the quadratic root for two numbers
2 import java.lang.Math;
3 class Quadratic
4 private static float Root1, Root2;
5 public static void main (String[] argv)
6     int X, Y, Z;
7     boolean ok;
8     int controlFlag = Integer.parseInt (argv[0]);
9     if (controlFlag == 1)
10    { X = Integer.parseInt (argv[1]);
11      Y = Integer.parseInt (argv[2]);
12      Z = Integer.parseInt (argv[3]); }
13    else
14    { X = 10;
15      Y = 9;
16      Z = 12; }
```

So, what we will end today's module with is, I will explain to you through one full code example how data flow coverage criteria works. So, here is the program that we will look at. So, this is a java program to compute the quadratic root of 2 numbers right. So, I assume that you have an equation of the form $Ax^2 + Bx + C$ right. And I want to be able to compute 2 values for x , the standard formula says that x is given as this formula right root 1 and root 2.

(Refer Slide Time: 26:14)

Example: Quadratic Root, contd.

```
23 // Three positive integers, finds quadratic root
24 private static boolean Root (int A, int B, int C)
25 {
26     double D;
27     boolean Result;
28     D = (double)(B*B)-(double)(4.0*A*C);
29     if (D < 0.0)
30     {
31         Result = false;
32         return (Result);
33     }
34     Root1 = (double) ((-B + Math.sqrt(D))/(2.0*A));
35     Root2 = (double) ((-B - Math.sqrt(D))/(2.0*A));
36     Result = true;
37     return (Result);
38 } // End method Root
39 } // End class Quadratic
```





So, how does the code do? The code has a class called quadratic there are 2 variables which are floating point numbers root1 and root2. What are x, y and z ? These are the coefficients of the polynomial, let us say $Ax^2 + Bx + C$, A, B and C are the coefficients. So, I pass them as x, y and z and then I have this Boolean entity called ok, which basically tells me whether this polynomial has a solution or not. And then what I do is I take these parameters. Suppose I do not pass them as normal integer coefficients then you just assign some default values in this case we have done 10, 9 and 12.

(Refer Slide Time: 26:58)

Example: Quadratic Root, contd.

```
17 ok = Root (X, Y, Z);
18 if (ok)
19     System.out.println("Quadratic roots:" + Root1, + Root2);
20 else
21     System.out.println ("No Solution");
22 }
```



And then you call this function method called root with these as parameters x, y and z.

So, root is the main method of this class quadratic that computes the quadratic root. So, root returns this value ok which says basically whether the quadratic roots have been computed or not and if they have been computed it is stored in the variables root 1 and root 2. So, if ok turns out to be true the values of root 1 and root 2 are printed. If ok turns out to be false the value is what is printed is to say that this equation has no solution. So, what is the focus here remember the focus here is to be able to look at interfaces. So, where is the interface the interface is at this statement, this is the call site where this main quadratic class calls a method root with 3 formal parameters x, y and z which are the coefficients of the quadratic equation that I am considering.

So, now we look at the code for root. What happens? It takes as input 3 positive integers, which are the coefficients and it finds the quadratic root. So, these are the three. So, x, y and z which are formal parameters here are passed as actual parameters A, B and C. So, what is coupled with what x is coupled with A, y is coupled with B and z is coupled with C right. And then it uses the standard formula that it computes for computing the quadratic equation. So, here what it computes is an intermediate value which is B squared minus 4 A C. You can look up for how to compute how to solve quadratic roots is a standard formula to solve an equation of the form $Ax^2 + Bx + C$. So, one, if I compute B squared minus 4 A C then one solution is given as $\frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$, that is what is this.

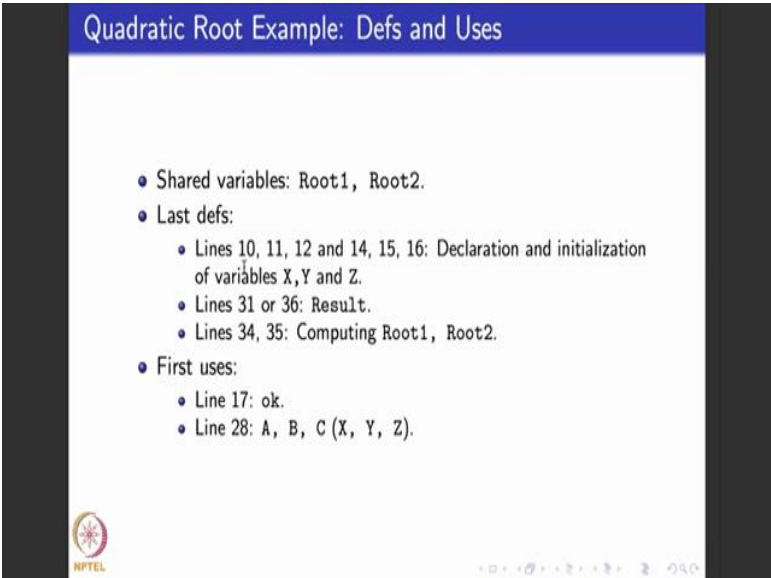
The next solution is given as $\frac{-B - \sqrt{B^2 - 4AC}}{2A}$. D computes B squared minus 4 A C. If D is 0 then there is no solution to the quadratic equation if D is less than 0, if D is greater than 0 then I can compute the 2 roots. So, what we do is we first compute B squared minus 4 A C, and check whether it is less than 0. If it is less than 0, then this procedure this method root stores the result is false and returns the result otherwise it computes this two square roots, stores it as root 1 and root 2 sets result to be true and returns the result.

So, how is the return passed here? Return is passed and assigned to this Boolean variable ok. If you remember ok was a Boolean variable. So, return is passed and assigned to this Boolean variable A. So, what if is true, if ok is true means what that I have computed root 1 and root 2 store it in these 2 variables as that is, how I return the result is true

across this statement, if ok was false then I have returned false which means I have not been able to compute this square root. So, the system correctly prints.

So, what is the goal now? The goal is there is this main class which calls is the method root. Root computes the, root basically checks if the given system of equations has a solution or not based on the value of D, and if it does not have a solution it says returns false. If it has a solution it outputs the two roots right. I want to be able to apply the data flow coverage criteria that I learnt now in this lecture to be able to test this. So, what are the things that we have to first do? We have to first identify the last defs and first uses.

(Refer Slide Time: 30:48)



Quadratic Root Example: Defs and Uses

- Shared variables: Root1, Root2.
- Last defs:
 - Lines 10, 11, 12 and 14, 15, 16: Declaration and initialization of variables X, Y and Z.
 - Lines 31 or 36: Result.
 - Lines 34, 35: Computing Root1, Root2.
- First uses:
 - Line 17: ok.
 - Line 28: A, B, C (X, Y, Z).

So, which are the shared variables? If you see root 1 and root 2 are the 2 shared variables. Which is the actual parameter that is passed and returned? X, y, z are the parameters that are passed, result is the parameter that is returned. So, which are the last defs? Last defs can occur at lines 10, 11, 12 or at lines 14, 15 and 16 which basically declare and initialize the 3 variables x, y and z. So, we will go back to the code. If you see lines 10, 11 and 12 take x, y and z as input. If it is some unacceptable value that is given as input it sets its own values for x, y and z as some fixed constants in line 14, 15 and 16.

After this it is directly passed to this called method root. So, the last defs for x, y and z are at lines 10, 11, 12 or at lines 14, 15 and 16. And when they are passed x, y and z is coupled with A, B and C respectively right. And then what else is passed, result. If you

see result is computed as false at line 31 or is true at line 36. So, it is passed here and then lines 34 and 35, if result happens to be true compute root 1 and root 2, lines 34 and 35 computes square root 1 and square root 2. Which are the first uses? Oh, result is returned as in line 17 right and x, y, z are passed A, B, C right we saw this.

(Refer Slide Time: 32:33)

Quadratic Root Example: Coupling du-pairs

Legend: Pairs of locations as (method name, variable name, statement number)

- (main(),X,10) — (Root(),A,28)
- (main(),Y,11) — (Root(),B,28)
- (main(),Z,12) — (Root(),C,28)
- (main(),X,14) — (Root(),A,28)
- (main(),Y,15) — (Root(),B,28)
- (main(),Z,16) — (Root(),C,28)
- (Root(),Root1,34) — (main(),Root1,19)
- (Root(),Root2,35) — (main(),Root2,19)
- (Root(),Result,31) — (main(),ok,18)
- (Root(),Result,36) — (main(),ok,18)

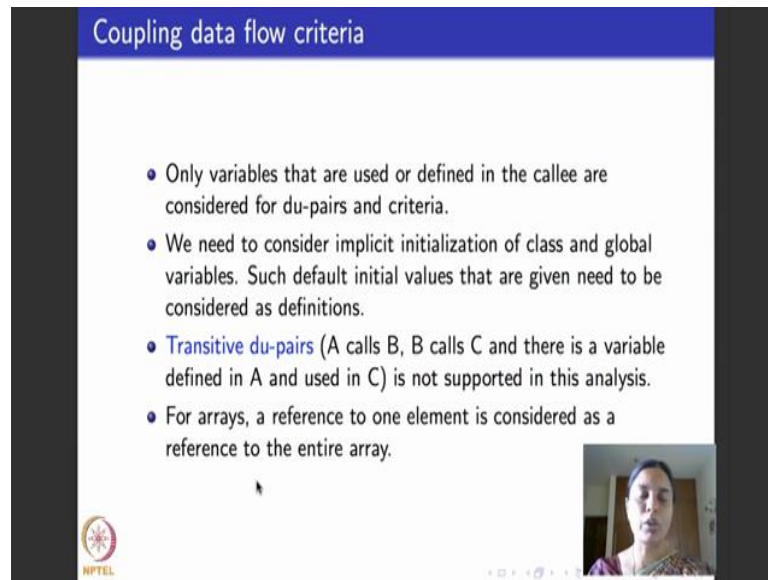
The slide also features the NPTEL logo in the bottom left corner and a small video feed of a presenter in the bottom right corner.

Now, what are the coupling D u pairs, as I told you X is coupled with A, Y is coupled with B, the Z is coupled with D. Main method calls the root at passes X, Y and Z, takes X, Y and Z as input in lines 10, 11, 12 or at lines 14, 15, 16. Wherever they are taken as input they are passed as A, B and C respectively at line 28. So, read this as, which is the method that is using it, which is the actual variable name, what is the statement number? That is what I given--- method, variable name and statement number. Method main calls method root at line number 10, by passing formal parameter as x, and actual parameter as where a is used at line number 28. And so on for the other things also. Method root method root returns back the value of root 1 to main which is used by main at line number 19. This must be the print statement yeah, this must be the print statement of the method main.

Similarly, for result which is coupled with the variable ok in the main variable right. Once I do this then I can normally write my data flow coverage criteria--- all defs, all coupling defs, all coupling uses, all coupling du-pairs. I have not written the TR and the test cases for data flow coverage criteria in this case because this particular example code

does not have any errors, but you can go ahead and do it as a small exercise for yourself. Write down the test requirement and give a test path that will satisfy the test requirement for coupling du-pairs coverage.

(Refer Slide Time: 34:21)



The slide is titled "Coupling data flow criteria" in a blue header. It contains a list of five bullet points. The third bullet point, "Transitive du-pairs", is highlighted in blue. In the bottom right corner, there is a small video inset showing a person speaking. The NPTEL logo is in the bottom left corner.

- Only variables that are used or defined in the callee are considered for du-pairs and criteria.
- We need to consider implicit initialization of class and global variables. Such default initial values that are given need to be considered as definitions.
- **Transitive du-pairs** (A calls B, B calls C and there is a variable defined in A and used in C) is not supported in this analysis.
- For arrays, a reference to one element is considered as a reference to the entire array.

Now, I hope this example would have clarified some details about how to use integration testing across parameter passing and shared variable interfaces to be able to test for data flow coverage criteria. Just a few other additional points that I would like to mention about data flow coverage criteria. One is when we look at data flow coverage criteria, even in the examples that I showed you, if you see I sort of did not give you the pieces of code that really did not matter to us. If you go back to this slide the rest of the code, I put it out as dot dot dot (...).

So why, because our focus is mainly on the values of the variables that are used or defined in the callee. We really do not look at other du-pairs when we are focusing on interface integration testing. The other thing we need to consider which this example didn't illustrate is that in languages like Java and C, there could be an implicit initialization of class or global variables. These implicit initializations will also have to be considered as definitions even if they do not come as explicit defs or statements in the code.


One more thing that I would like you all to note is that suppose you had this kind of a situation. A method A calls a method B, B in turn call C and let us say there is a variable

that is defined in A and used in C. So, this is what is called a transitive du-pair right. It goes down a callee hierarchy. The kind of analysis that I have explained to you in today's lecture will not suffice to do this kind of call graphs.

So, you need slightly different kinds of analysis for that and that we will not deal with in the scope of this course. So, if you have to do this kind of analysis, I would say that you change the code a little bit for to make method A use method B's variable directly coupled them with and B's and C's variable, you couple it. Avoid coupling A and C variable I will point you to some papers at the end of this lecture where you can read up some more information about this, but the kind of analysis testing that we did today will not work for transitive pairs.

And the other thing is that whenever you look at arrays, if I pass if I have to pass an array as a formal parameter or a natural parameter, how will I give ? There the interpretation that we take is a reference to one element in the array is considered as the reference to an entire array.

(Refer Slide Time: 36:49)



The slide is titled "Coupling data flow criteria and certification standards" in a blue header. It contains two bullet points: "Several certification standards for certifying embedded software insist on testing using coupling criteria." and "Example: DO-178C used by Federal Aviation Authority (FAA) in the USA states that 'Analysis should confirm the data coupling and control coupling between the code components' (page 33, section 6.4.4.2)." In the bottom right corner, there is a small video inset showing a woman speaking. The NPTEL logo is in the bottom left corner.

- Several certification standards for certifying embedded software insist on testing using coupling criteria.
- Example: DO-178C used by Federal Aviation Authority (FAA) in the USA states that "Analysis should confirm the data coupling and control coupling between the code components" (page 33, section 6.4.4.2).

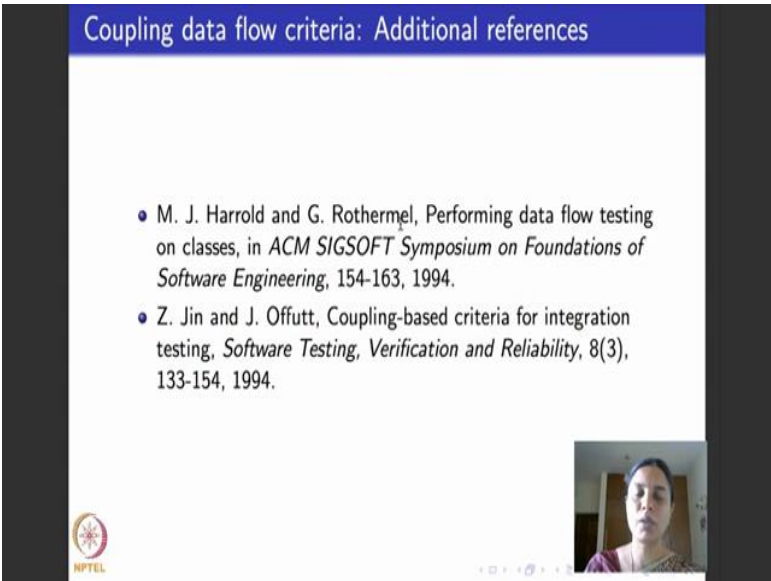
I would also like to mention one important point. Data flow kind of testing that we saw today which is coupling data flow criteria is very important when it comes to certifying testing safety critical software. If you remember right in the first week I told you that there are several standards and several standards that test for safety critical software. What are safety critical software? Software that flies our planes drives our cars controls

nuclear power plants maybe the software that runs metro trains across the metros of India, because if there is a failure in this software it can mean life threatening loss or damage.

So, these safety critical software go through certification standards which is what we discussed in the first week. One example of the certification standard that explicitly mentions coverage criteria is the standard DO-178C. It is used by federal aviation authority in the USA and in India also several of the DRDO firms used 178C as the reference for testing avionics applications. In that if you go to the standard look at page 33, this particular section it explicitly mentions 2 tests for coupling data flow analysis which is what is the kind of testing that we looked at today.

So, it says analysis should confirm the data coupling and control coupling between the core concept. So, it asks you to do testing and provide evidence of the fact that you have tested the interface for both structural coverage and data coupling.

(Refer Slide Time: 38:21)

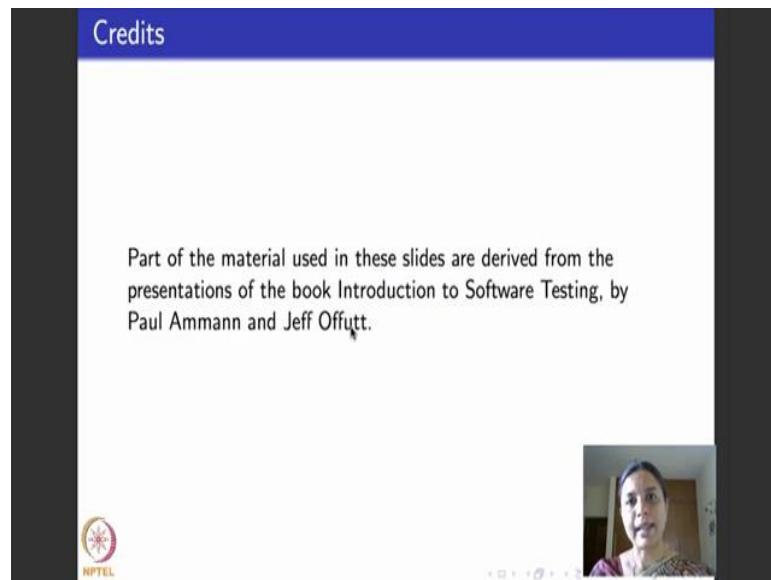


The slide is titled "Coupling data flow criteria: Additional references" in a blue header. It contains two bullet points listing references. In the bottom right corner, there is a small video feed of a person. The NPTEL logo is in the bottom left corner.

- M. J. Harrold and G. Rothermel, Performing data flow testing on classes, in *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 154-163, 1994.
- Z. Jin and J. Offutt, Coupling-based criteria for integration testing, *Software Testing, Verification and Reliability*, 8(3), 133-154, 1994.

So, here are some papers where you can read up for more information. This paper by Harrold and Rothermel is considered the first definition, first place where this coupling and data flow testing across coupling was introduced. And this paper by Jin and Offutt is the paper that you should refer to for explicit first time introduction of the term last def and first uses; thank you, what we will do next module is to be able to look at specifications and see how graphs are used for specifications.

(Refer Slide Time: 38:45)



Thank you.

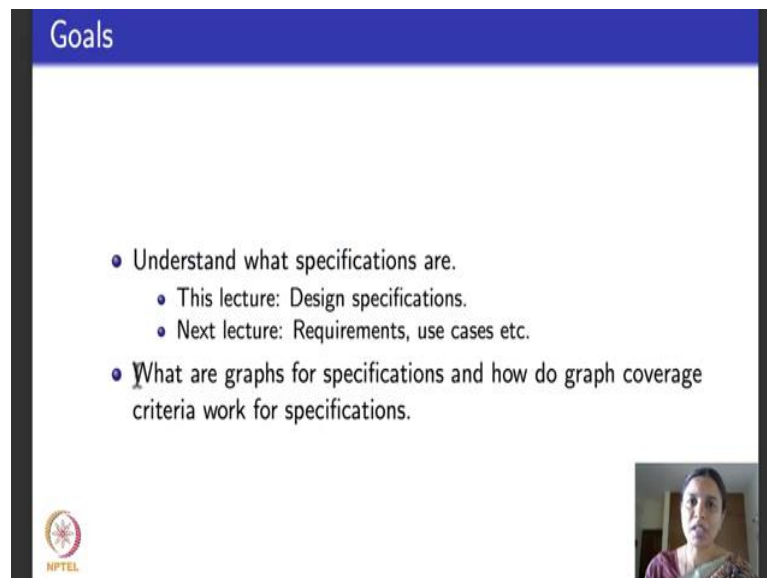
Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 18
Specification Testing and Graph Coverage

Hello again, we are now doing the fourth lecture of fourth week. As I said this will be the last week where we look at graph coverage criteria. We saw graph coverage criteria on code. Last 2 lectures I told you how to apply graph coverage criteria for integration testing that deals with testing for design that is interaction and interfaces between modules. Now we will move on to the requirements on design.

So, what we will see in this module and in the next module is somewhere between design and specifications and see how graph criteria apply to test these also. So, we will understand what specifications are. The focus of this lecture would be specifications that actually talk about design itself.

(Refer Slide Time: 00:51)

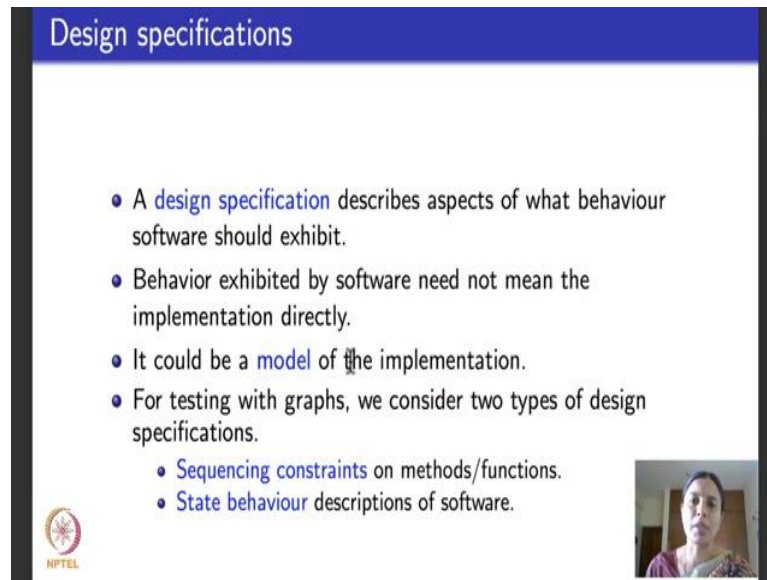


The slide has a blue header with the word "Goals" in white. Below the header, there is a white area containing a bulleted list of goals. In the bottom right corner of the slide, there is a small video inset showing a woman (Prof. Meenakshi D'Souza) speaking. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.

- Understand what specifications are.
 - This lecture: Design specifications.
 - Next lecture: Requirements, use cases etc.
- What are graphs for specifications and how do graph coverage criteria work for specifications.

So, I will explain to you through examples what it means to write specifications that talk about design because there are several specifications that we need to write along with the design to ensure the things are working correct, even though these specifications may not explicitly map to the requirements of a software.

(Refer Slide Time: 01:35)



The slide is titled "Design specifications" in a blue header. It contains a bulleted list of points. In the bottom left corner, there is a small circular logo with the text "HPTCL" below it. In the bottom right corner, there is a small video inset showing a person's face.

- A **design specification** describes aspects of what behaviour software should exhibit.
- Behavior exhibited by software need not mean the implementation directly.
- It could be a **model** of the implementation.
- For testing with graphs, we consider two types of design specifications.
 - **Sequencing constraints** on methods/functions.
 - **State behaviour** descriptions of software.




So, today we will deal with those kind of specifications and in the next lecture, I will introduce you to finite state machines, finite state automata which are a popular model to write requirements and then we will see how graph coverage criteria work on them. What is a design specification? So, as I told you design describes; what are the modules in the software, how software is broken up into its components and how they interact with each other and what happens? A design specification talks about what are the requirements or constraints that need to be satisfied by the design that is being written and base-lined to make the design correct.

Not all design specifications are derived directly from the requirements of software. Some of them are extra specifications written to ensure that the design is indeed correct. So, design specification can be thought of as the path of the model, design model corresponding to the implementation. While we do graph based testing we will consider 2 types of design specifications. What I will do in today's lecture would be what is called sequencing constraints which are constraints on methods and functions that come in the design or modules that come in a design. Not all sequencing constraints can be used to write all kinds of design specifications. So, certain things that cannot be used we will look at finite state machines that describe state behavior that we will do in the next week.

(Refer Slide Time: 02:52)

Sequencing constraints

- Sequencing constraints are rules that impose constraints on the order in which methods may be called.
- They can be encoded as preconditions or other specifications.
- They may or may not be given as a part of the specification or design.
- Testers need to derive them if they don't exist— they are considered another rich source of errors.

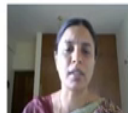




So, what is the sequencing constraint? Before I introduce you to what is a sequencing constraint, I would like to begin with a small example.

(Refer Slide Time: 02:54)

Sequencing constraints: An example

```
public int dequeue()
{
    // Pre: At least one element must be on the queue.
    ...
    ...
    public enqueue(int e)
    {
        // Post: e is on the end of the queue.
        ...
        ...
    }
}
```



Page 5 of 17

Let us say this is just another abstract snippet of some design it says that there are 2 methods here. Both the methods deal with the queue data type, you know what a queue data type is right? A queue is a first in first out list or an array of elements. So, what are the typical operations that come with a queue? The typical operations that come with a queue are enqueue and dequeue - enqueue inserts an element into the queue dequeue

removes an element that is already present at the other end of the queue. So, there are these 2 methods - one is `deQueue` which is removed the element from the queue and assign it to an integer variable the other is insert an integer into a queue right. And I have not really given the code for these methods, but if you see there are 2 comments that are written here, for `deQueue` there is a comment which is named `Pre` which says at least one element must be on the queue.

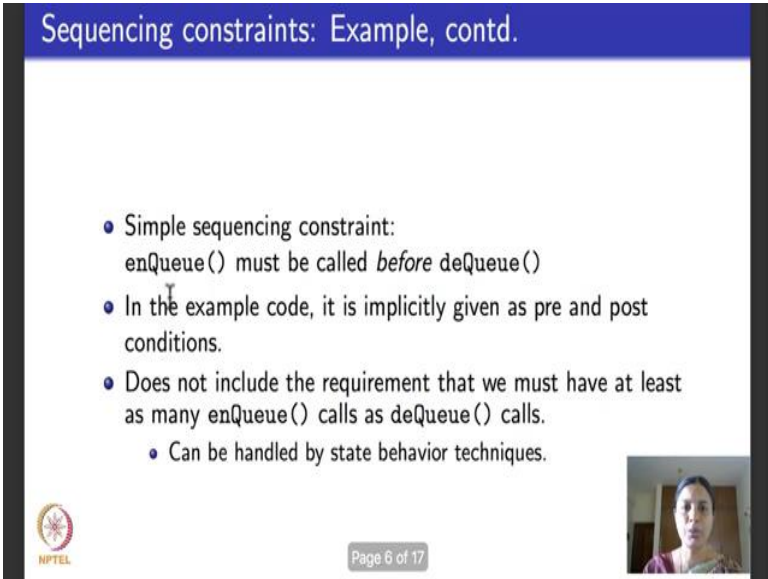
So, you can read this as, it is a precondition which says that there must be at least one element in the queue. Why is this thought of as a precondition? Let us say the queue was empty. Then how do I `deQueue` anything. I will not be able to `deQueue`, `deQueue` will give me an error. So, to be able to `deQueue` I need at least one element in the queue. So, I state it like a precondition and if you read what is written in `enQueue`, I have written again as a comment the first condition which says that `e` is at the end of the queue. So, this is which basically says that the queue is like a FIFO data structure, first in first out data structure. So, when I insert an element in the queue it goes to the end of the queue and for me to remove an element from the queue the precondition is that there must be at least one element in the queue.

So, these can be thought of as basic constraints that tell you what methods `enQueue` and `deQueue` should satisfy. The methods `enQueue` and `deQueue` could be used as a part of a mainstream functionality that is meant to cater to a particular requirement of a software. But wherever they are used these preconditions and post conditions basically say what will happen or what should happen when they are used. Precondition for `deQueue` says that the queue should not be empty. So, there should have been at least one `enQueue` operation without another `deQueue` operation before the current `deQueue` operation.

And the post condition for `enQueue` says that after an element is inserted into the queue it goes to the end of the queue. So, these are what are called sequencing constraints. So, what is a sequencing constraint? These can be thought of as the rules that impose some constraints in the order in which the methods in a particular piece of code can be called. Like for example, for me to be able to do `deQueue` I should have done at least one `enQueue` in the past. So, these can be written as preconditions, post conditions, asserts or any other way of writing it and they are requirements that we impose on the design of the software.

On other thing that I told you that has to be noted is that these are typically not given as a part of the specification document or as a part of the requirement document. These are written along with design typically by designers to ensure that their design works correctly in the sense of it is necessary that these constraints are satisfied for the design to be able to meet its expected functionality. Suppose they are typically not given then the onus is on the tester to be able to derive them and test them.

(Refer Slide Time: 06:46)



Sequencing constraints: Example, contd.

- Simple sequencing constraint:
enQueue() must be called *before* deQueue()
- In the example code, it is implicitly given as pre and post conditions.
- Does not include the requirement that we must have at least as many enQueue() calls as deQueue() calls.
 - Can be handled by state behavior techniques.

NPTEL

Page 6 of 17

So, this is the queue example that we saw of a simple sequencing constraint on a queue which basically says that enQueue must be called before a deQueue. If that is not the case then the queue could be empty because it begins with an empty queue and there is nothing to deQueue from right. So, I do not want to keep trying to deQueue from an empty queue. So, insist that there must be at least one enQueue operation before a deQueue operation.

Here in this example, such a sequencing constraint is given implicitly as pre and post conditions. In fact, you can write several such other sequencing constraints for the queue example itself. Another simple sequencing constraint is something like this--- it says that there must be at least as many enQueue calls as there are deQueue calls. Why do we need this? Let us say I have a queue the queue contains at any point in time at some point in time let us say it contains 3 elements. So, what can I do? I can do continuously 3 deQueues. So, how did the three elements come into the queue? There must have been 3

enQueue operations in the beginning before the 3 deQueue operations. It can never be the case that there were only 2 enQueue operations, if there were only 2 enQueue operations in this queue then the queue cannot have three elements right assuming that the queue is empty to start with.



So, what it says is that to be able to deQueue I should have enQueued in element in the past. So, at any point in time the following hold as an invariant: the number of enQueue operations should be greater than or equal to the number of deQueue operations. So, if you see the word number matters here; we are talking about the number of enQueue operations and the number of deQueue operations. Whenever I am counting the number of operations related to a particular piece of software execution I typically you cannot write it as a simple assertion or a sequencing constraint. I need what are called state based models for that. We will not look at state based models in this lecture that will be the focus of next lecture. But what I wanted to tell you as a part of this lecture is that when I take a queue example or any other kind of example which is meant to be like a data structures serving as a part of a method, I could write constraints which talk about how the data structure should work correctly such constraints are called sequencing constraints.

Some sequencing constraints can be written as simple invariants like we saw here enQueue must be called before a deQueue. Some sequencing constraints need to deal with history or memory or number and they need state machines to be able to write them which we will see in the next lecture.

(Refer Slide Time: 09:20)

Testing sequencing constraints

- Sequencing constraints may or may not be given explicitly, might not be given at all.
- Absence of sequencing constraints usually indicates more faults.
- Tests are created as sequences of method calls, testing if the sequence obeys the constraints.
 - Usually write tests to find errors in constraints or missing constraints.



So, as I told you sequencing constraints typically are not explicitly given say they always have to be written by the designer and it could very well be the case to the designers mis-writing them if the designers miss writing them then the onus is on the tester to be able to identify, write them and test for all these sequencing constraints to be met. Like interfaces, when you go into integration testing absence of correct sequencing constraints or a software not adequately tested for correct sequencing constraints is meant to lead to a rich source of errors and onus is on testers to be able to identify and eliminate these errors during integration testing.



(Refer Slide Time: 10:01)

Testing sequencing constraints: An ADT example

Consider a class FileADT that encapsulates operations on a file.
Class FileADT has three methods:

- open(String fName): Opens file with name fName.
- close(): Closes the file and makes it unavailable.
- write(String textLine): Writes a line of text to the file.

What are natural sequencing constraints you would expect for this class?

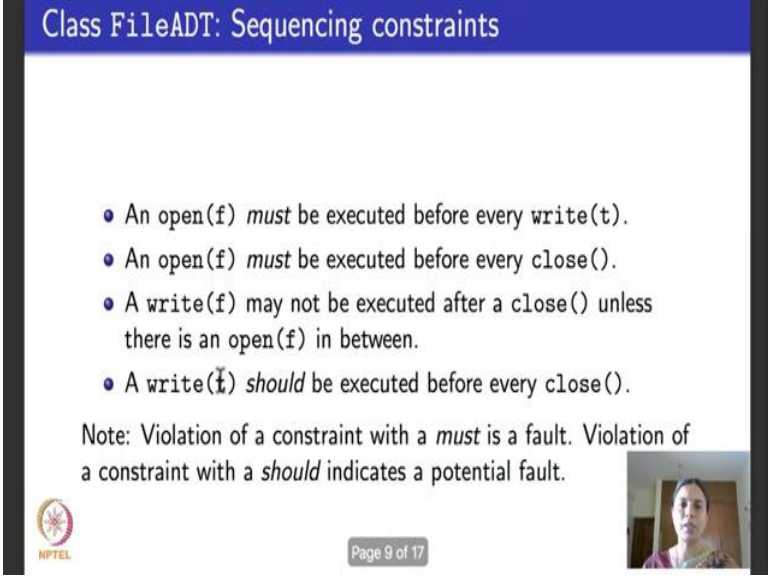


So, we will see how to test for sequencing constraint through an example. Consider a class called file ADT, ADT stands for abstract data type. This is the class that encapsulate operations on a file, that file has some kind of data. What is the typical bare minimum set of operations that you would do on a file? You could open a file which is done through this command: open file name with the name fName. So, this is a method open which let us you to open a file name with the name fName and once a file is opened you could close it. I have not given an argument as the name of the file here just for convenience sake, but you could as well write close a string followed by file name. So, that you know which file to close because I have not written an argument for close here I assume through the examples that we look at that I am working with only one file at a time. Suppose it happens to be the case that you are working with more than one file you need to give close operations for that file name also.

And once the file is open before being closed you would want to work with a file you could read data from the file or you could write on to a file. So, here is a method that writes on to a file. So, this method writes a line of text called text line which is a string to the file.

So, there is a class file abstract data type in short, file ADT which encapsulates three methods open file, close file and write data to a file. So, if you think about sequencing constraints for any code that uses these methods. So, there is a code that deals with some file operations and in the process of dealing with those file operations it uses methods that are available as a part of this class. What would be sequencing constraints that you will write for such kind of code? The obvious kinds of sequencing constraints are as follows.

(Refer Slide Time: 11:55)



Class FileADT: Sequencing constraints

- An `open(f)` *must* be executed before every `write(t)`.
- An `open(f)` *must* be executed before every `close()`.
- A `write(f)` may not be executed after a `close()` unless there is an `open(f)` in between.
- A `write(t)` *should* be executed before every `close()`.

Note: Violation of a constraint with a *must* is a fault. Violation of a constraint with a *should* indicates a potential fault.

Page 9 of 17

A file must be opened before you can write on to it, an open file must be closed before you finish operations, better not leave it open. A write may not be executed after a close unless there is an open in between. So, what it says is suppose you open a file and then you write and then you close the file and then you have to open it once again to be able to write. So, once you close the file you cannot write on to it without opening it again and then the fourth sequencing constraint says that a write should be executed before every close.

So, what is the fourth one say? Fourth one says write should be executed before every close. So suppose this is violated then what will happen then it means that a piece of software is trying to do some dummy operations on a file, it could just open a file close a file, open a file, close a file without doing anything. Even it does not read from a file, it does not write on to a file. So, it is just keeping itself busy, but doing nothing. So, we would want to prevent such things. So, we say that once a file is open do something with it, do something like a write with it before you close it.

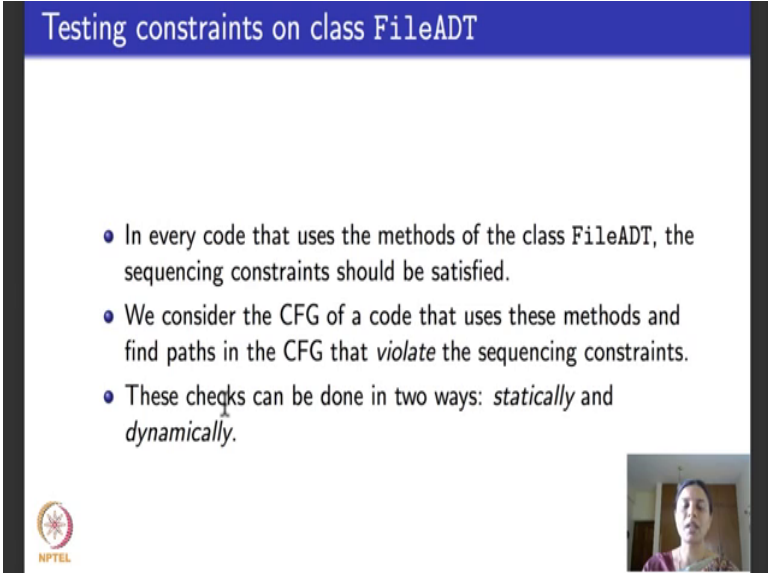
The other thing that you must notice in this slide is that I have put words like *must* in the first 2 constraints and the word *should* in the fourth constraint in italic font, what do we mean by that? Typically when we write requirements for specifications or constraints you will encounter these words a lot when you write requirements in English: 'may', 'must', 'shall', 'should' and what do they usually mean? Usually the presence of a word

must means that if that requirement or constraint is violated then there is a fault for sure and the word like 'should' should be interpreted as if that requirement or constraint is violated then there might be a fault. So, it is clear even for this example that something like that holds.

Suppose, let us say the second requirement consider second constraint it says open f must be executed before every close. So, suppose this is violated which means what somebody has tried to close a file that was not even open, that is not possible so it will definitely result in a fault and it is a must sequencing constraint. Look at the fourth sequencing constraint, it has the word 'should', should means that somebody is trying to write on to a file I mean somebody is opened the file and closed the file without writing on to it. It is harmless in the sense that it does not really cause the serious fault, but it is useless also. So, a requirement with a 'should' may or may not result in a fault. So, we have to be careful about when we use must and should kind of requirements, these are typically done by requirement engineers who write software requirement or by architectures specify the design.

Now, I have class that contains the three methods.

(Refer Slide Time: 15:09)



The slide has a blue header with the text "Testing constraints on class FileADT". Below the header, there are three bullet points:

- In every code that uses the methods of the class FileADT, the sequencing constraints should be satisfied.
- We consider the CFG of a code that uses these methods and find paths in the CFG that *violate* the sequencing constraints.
- These checks can be done in two ways: *statically* and *dynamically*.

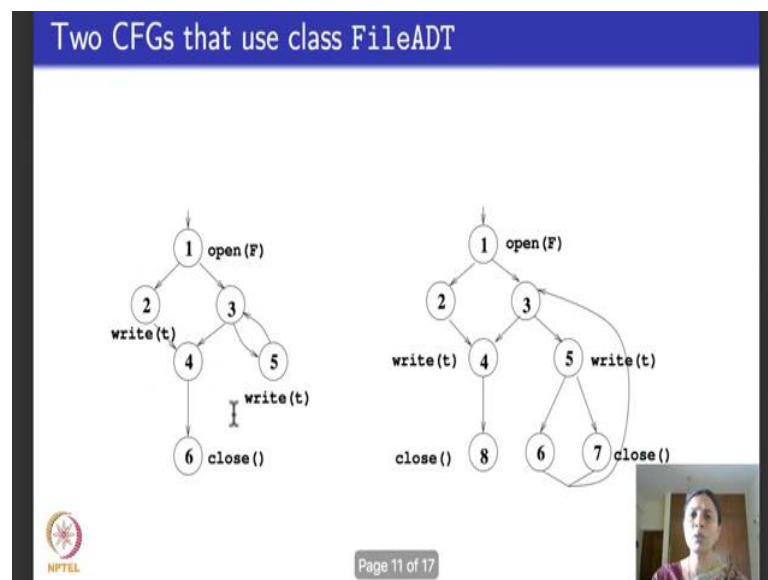
In the bottom left corner, there is a small circular logo with the text "NPTEL" below it. In the bottom right corner, there is a small video inset showing a person speaking.

These are the sequencing constraints on the method. Now what I want to do is the following. Consider a piece of code that uses methods from this class, then that piece of code should satisfy all the sequencing constraints that we have written right, which

means it must open a file before it writes on to it and promptly close the file and it must not ideally open and close a file without doing a write operation in between. Those are the kind of things that we do. So, what we do is that we consider the code that uses the methods from this class you consider the control flow graph of that code, then what will be your test requirement and test paths? They would be test paths in the control flow graph that check if any of these constraints are violated.

Typically if sequencing constraints there are two ways of doing it, you can try to violate the sequencing constraints in a static way or you could try to violate the sequencing constraints in a dynamic way.

(Refer Slide Time: 16:08)



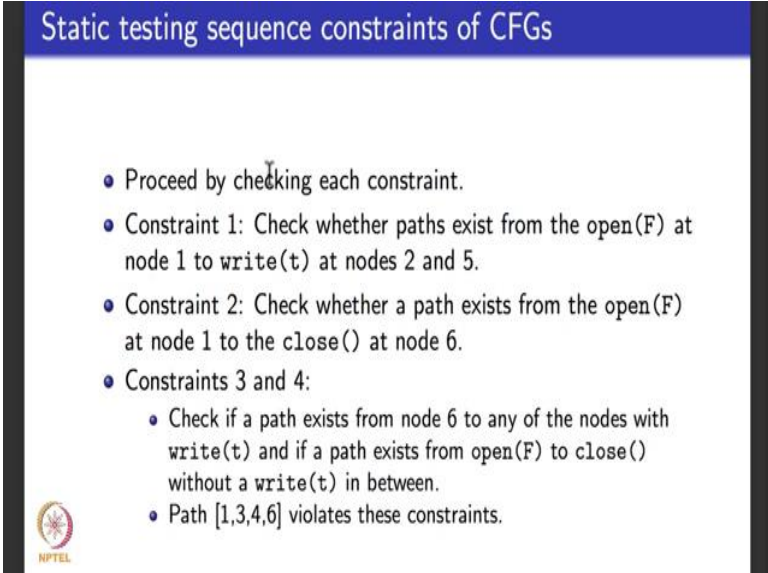
So, we first look at static way. So, before we move on to that here are 2 examples of 2 different pieces of code that use the methods of the class file ADT. On the left hand side it is the CFG corresponding to one piece of code that uses methods in a particular way. On the right hand side this CFG corresponding to another piece of code that uses a method in another particular way. So, let us focus on the CFG in the left hand side at node 1 it calls the method open F which opens the file F and node 2 it writes data on to that file, at node 3 it goes into a loop where it repeatedly writes data to the file several times and then when it comes out it closes the file. This is the control flow graph corresponding to some code that uses these methods. I have again abstracted out and

given you only the details of the CFG as relevant to this lecture, in between it could do several other operations we have not depicted that in the CFG.

So, here is a CFG of another graph that uses the same methods from the class file ADT. Always it begins with open; please remember that unless it begins with open if it directly does a write or a close that will violate the sequencing, one of sequencing constraints. So, it always begins with the open and then like the earlier code, it can do one write and close the file or it can write and close several times in a loop which is this right hand side 3, 5, 7 and get back.


So, now what we will do is that if this is the first CFG of some code, this is a second CFG of some of the code, we will take these sequencing constraints and see whether these two pieces of code or CFGs that uses these methods do they satisfy the sequencing constraints or do they violate the sequencing constraints? As I told you this can be done in two ways: statically or dynamically. What I mean by statically? Statically means I do not execute the code. I consider the CFG as a graph and then try to see if sequencing constraints are violated. Typically static analysis is not considered a path of testing, but I will just tell you for the sake of completeness here.

(Refer Slide Time: 18:34)



Static testing sequence constraints of CFGs

- Proceed by checking each constraint.
- Constraint 1: Check whether paths exist from the open(F) at node 1 to write(t) at nodes 2 and 5.
- Constraint 2: Check whether a path exists from the open(F) at node 1 to the close() at node 6.
- Constraints 3 and 4:
 - Check if a path exists from node 6 to any of the nodes with write(t) and if a path exists from open(F) to close() without a write(t) in between.
 - Path [1,3,4,6] violates these constraints.

 NPTEL

So, how the static testing of sequence constraints a CFG work? It proceeds by checking one constraint after the other. Go back to the constraint slide, there are four constraints - first says open must be executed before write, second says open must be executed before

close, third says write cannot be executed after the close unless there is an open in between, fourth says that write should be executed before every close.

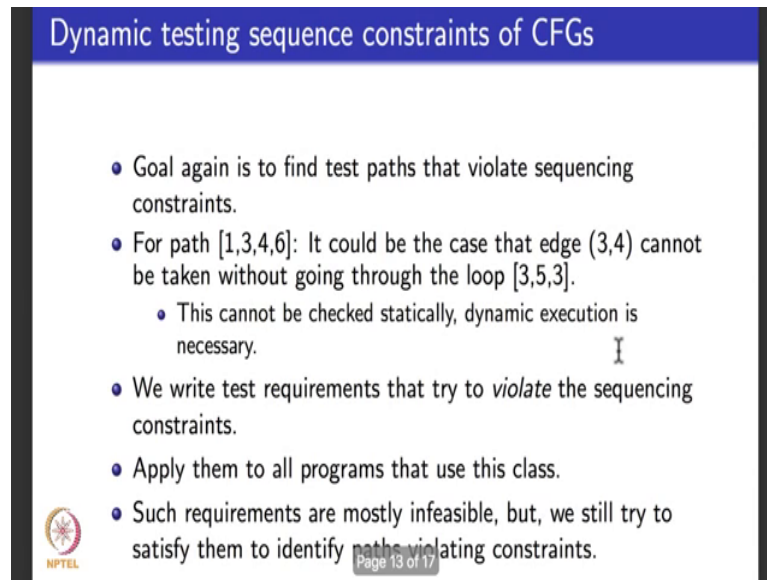
So for constraint one, we check if there is a path from an open to a write, where is open here I am focusing only on the CFG on the left hand side for the purposes of this, open is at node 1, there are writes at node 2 and node 5. So, for the first constraint I check whether there is a path from the open at node 1 to writes at node 2 and node 5. It so happens then there are, so we doing fine as far as the first sequencing constraint is concerned.

For the second one similarly you check whether there is a path from an open at node 1 to the close and node 6. It so happens that there are, if you see 1, 2, 4, 6 is one path, 1 2 3 4 is one path, 1, 2, 3, 5, 3, 4, 6, this one path. So, there are paths, so as far as the second requirement open must be executed before every close I am still doing fine.

For constraints 3 and 4, I have to check if there is a path from node 6 to any of the nodes with a write and if there is a path from open to close without write in between. So, there is a path from node 6 which is a close to any of the writes. So, if you see there is a write that happens at node 5 before close, there is a write that happens at node 2 before the close. But if you see there is a path from open at node 1 to close at node 6 without a write. Which is that path? That path goes through node 3, I open the file I go to node 3, I do not do anything specifically I do not do a write then I go to four there I skipped the loop that is and then I close. So, I have open the file and close the file without doing a write in between. So, which is the constraint that it violates? It violates the fourth constraint, it violates the constraint that the write should have been executed before a close, that is what is given here. Path 1, 3, 4, 6 violates the constraint.

Please note that in this slide all the constraints we are reasoning about with reference to this CFG, the CFG on the left hand side. The CFG example of the right hand side I will use it when I do dynamic testing.

(Refer Slide Time: 21:10)



Dynamic testing sequence constraints of CFGs

- Goal again is to find test paths that violate sequencing constraints.
- For path [1,3,4,6]: It could be the case that edge (3,4) cannot be taken without going through the loop [3,5,3].
 - This cannot be checked statically, dynamic execution is necessary.
- We write test requirements that try to *violate* the sequencing constraints.
- Apply them to all programs that use this class.
- Such requirements are mostly infeasible, but, we still try to satisfy them to identify ~~paths violating~~ constraints.

HPTEL

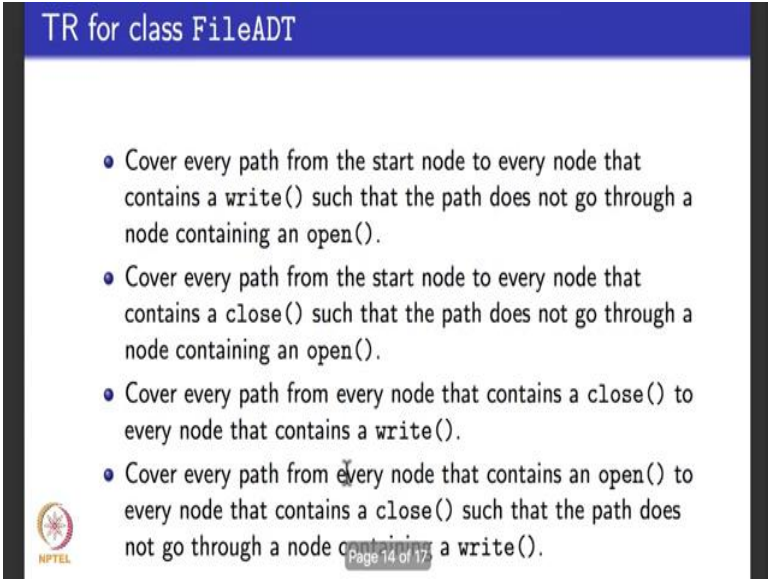
Page 13 of 17

Now when it comes to dynamic testing my goal is somehow to be able to find paths that violate sequencing constraints. So, I write test requirements or TRs that tell you what to cover. So, I will say cover for sequencing constraint, second sequencing constraint and so on for each of the sequencing constraint and my goal is to find test paths that violate the satisfaction of the sequencing constraint, right. So, if I consider dynamic testing and I go back to the first CFG the path 1, 3, 4, 6 which potentially violated constraint 4 could not be realistic.

If you remember I told you whenever we look at control flow graphs, it many times may not be feasible to write a path that all together skips a loop. 1, 3, 4, 6 in this CFG is such a path. It could be the case the loop was a do while loop and you are forced to execute one iteration of the loop at least to be able to do. So, suppose you have that then when you do dynamic testing you will know about it because you actually execute the software and then path 1, 3, 4, 6, might not be an actual violation of a sequencing constraint, but in static testing you will flag it as a potential violation of the fourth sequencing constraint, but when you do dynamic testing you will know whether it was actually a false positive or it is a genuine violation of a sequencing constraint. It will be a false positive if the code for the CFG forces you to enter the loop it will not be a genuine violation it would be a false positive, but if the code allows you to skip the loop then it will be a genuine violation of the fourth constraint. You can find out this through dynamic testing.

So, when the other thing that we do in dynamic testing as I told you is to write test requirements that violate the sequencing constraint and apply these test requirements to all the programs that use the methods from this class right. Typically for large programs is actually quite difficult to write all the sequencing constraints, to write the TRs for each of the sequencing constraints and to be able to get tests pass and its usually undecidable, but we will not really worry about proving which are the programs for which it is decidable or undecidable, the idea is to be able to understand how it is done.

(Refer Slide Time: 23:41)



TR for class FileADT

- Cover every path from the start node to every node that contains a write() such that the path does not go through a node containing an open().
- Cover every path from the start node to every node that contains a close() such that the path does not go through a node containing an open().
- Cover every path from every node that contains a close() to every node that contains a write().
- Cover every path from every node that contains an open() to every node that contains a close() such that the path does not go through a node containing a write().

Page 14 of 17

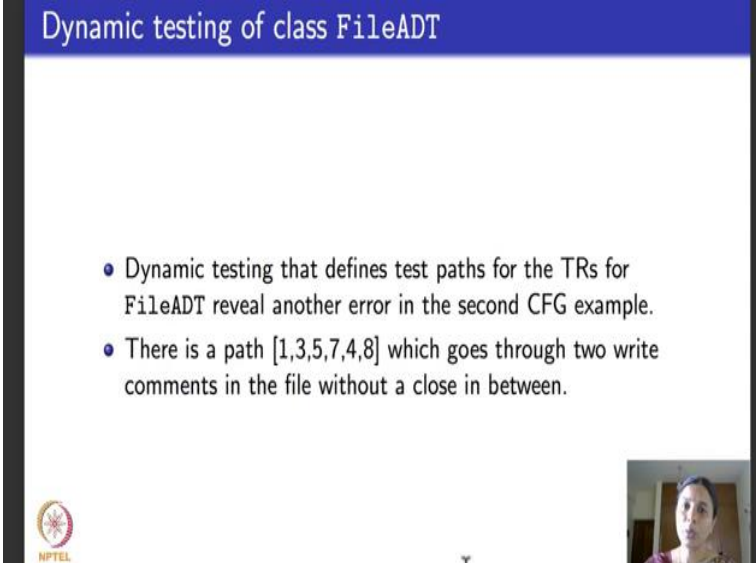
So, what I do now is I write test requirements for all the constraints this one TR for every constraint, there were four constraints so there are four TRs. First one says that you cover every path from the initial node to every node that contains the write such that the path does not go through an open, this will violate the first constraint right. So, it says you cover every path that goes through a write, but without going through an open. Let us go back and see what constraint one said open must be executed before every write. Suppose you are able to find a path that will meet this TR then you have violated the first constraint.

Similarly, for second constraint, second constraint says that before a close a file must be open. So, the test requirement to violate the second constraint will say you cover every path from the start node to every node that contains the close command call to the close method such that the path does not go through an open method. Suppose you find the test

path you have successfully found a test path that violates the second constraint. Similarly for the third sequencing constraint your TR says you cover every path from every node that contains a close to the path the node that contains a write which means the write is happened after a close. Fourth TR says cover every path from the node that contains an open to a close without going through a write.

I have basically written the same sequencing constraints, but in terms of coverage criteria that violate the sequencing constraints, I hope this is clear.

(Refer Slide Time: 24:25)



The slide has a blue header with the text "Dynamic testing of class FileADT". Below the header, there are two bullet points:

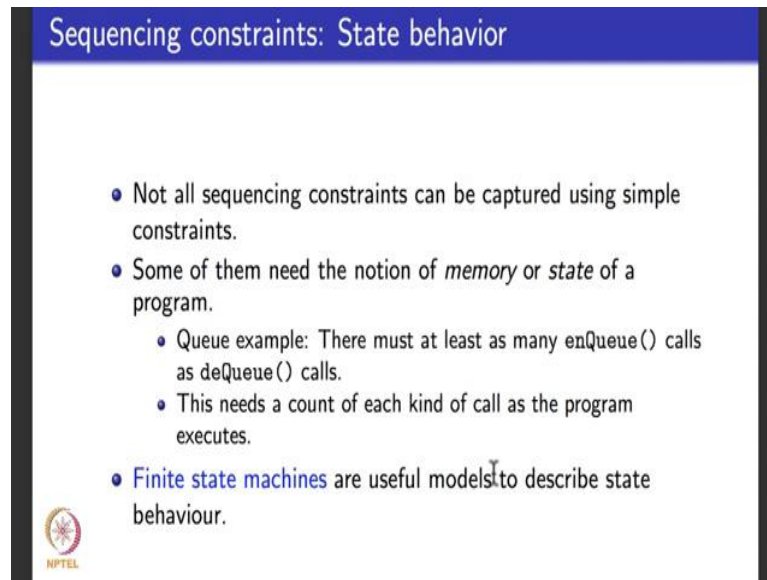
- Dynamic testing that defines test paths for the TRs for FileADT reveal another error in the second CFG example.
- There is a path [1,3,5,7,4,8] which goes through two write comments in the file without a close in between.

In the bottom left corner, there is a logo for NPTEL. In the bottom right corner, there is a small video inset showing a person speaking.

So, if I do dynamic testing I will find that again the fourth constraint is violated. So, in this case we will look at the second CFG this one. How is the fourth constraint violated? There is this path, 1, 3, 5, 6, 3 which tries to write and not close a file. It tries to write once, goes back to 3 tries to write once more and does not close the file and after that only closes a file.

So, through dynamic testing I will be able to find out whether there is a violation of the sequencing constraint or not.

(Refer Slide Time: 26:08)



The slide has a blue header with the text "Sequencing constraints: State behavior". Below the header, there is a white area containing a bulleted list. The list starts with "Not all sequencing constraints can be captured using simple constraints." followed by "Some of them need the notion of *memory* or *state* of a program." which is further indented with two sub-points: "Queue example: There must at least as many enqueue() calls as dequeue() calls." and "This needs a count of each kind of call as the program executes." The final bullet point is "Finite state machines are useful models to describe state behaviour." In the bottom left corner of the slide, there is a circular logo with a star and the text "HPTEL" below it.

- Not all sequencing constraints can be captured using simple constraints.
- Some of them need the notion of *memory* or *state* of a program.
 - Queue example: There must at least as many enqueue() calls as dequeue() calls.
 - This needs a count of each kind of call as the program executes.
- Finite state machines are useful models to describe state behaviour.

So, in summary, what do I do? I identify sequencing constraints if they have not been documented by designers. What is my test requirement? I write test paths that try to cover in the test cases in such a way that if my test path is found then one of the sequencing constraints is violated. So, I write one test coverage criteria for violating each of the sequencing constraints and suppose I identify in a test path that achieves this coverage criteria then I have successfully found a path in the graph that violates the sequencing constraint which leads to an error.

Sequencing constraints can be of two types: the first type this plain example like we saw for the queue example or the file open, file close example. Sometimes you can have sequencing constraints that need to count the numbers that happened in the past, the number of certain kinds of operations that happened in the past. Those are called state full sequencing constraints. And in the next module will introduce finite state machines or finite state automata with which we can work to model state behavior related sequencing constraints.

Thank you.


Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 19
Graph coverage and finite state machines

Hello there, this is the last lecture of the fourth week, we will be winding up with looking at graphs today. This will be the last lecture where we will see testing based on coverage graph coverage criteria for now, what I will do today is introduce you to this popular model called finite state machines most of you might be familiar with it, and see what kind of graphs they are and how graph coverage criteria will be applicable to them.

So, far what did we see now? We saw graphs structural coverage criteria, data flow coverage criteria, then we saw how to apply to source code then we saw how to apply them to data flow along with source code and the CFG. Then we saw how to apply them to design elements specifically we saw how to apply it to sequencing constraints; and then I told you some amount of sequencing constraints need you to keep track of state information. So, when you need to keep track of state information, finite state machines or finite state automata come in to be very handy.

(Refer Slide Time: 01:10)



The slide is titled "Goals" in a blue header. It contains two bullet points:

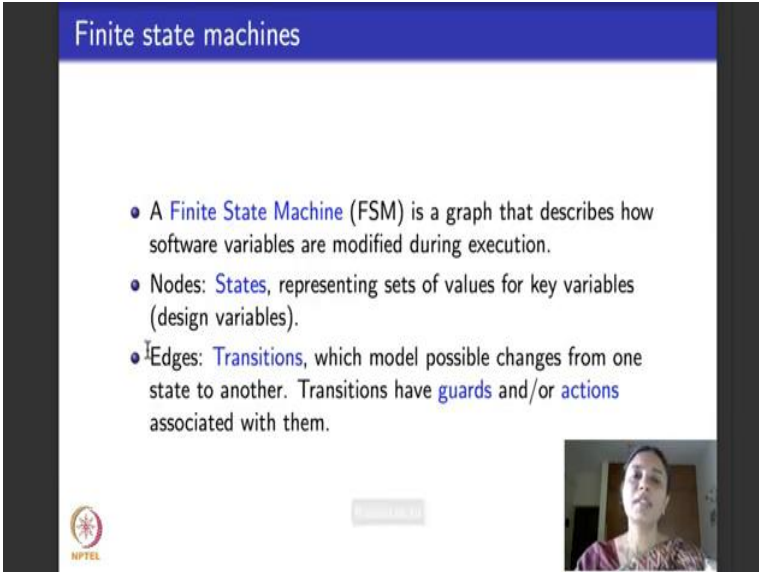
- Understand what finite state machines are and how they model design specifications.
- What do graph coverage criteria over finite state machines test about specifications?

In the bottom right corner, there is a small video inset showing Prof. Meenakshi D'Souza. In the bottom left corner, there is a logo for NPTEL.

So, today we will introduce those models and see how a graph coverage criterion is useful for them. Another popular design models are what are called UML diagrams.

UML is basically Unified Modeling Language, a collection of popular diagrams each of which is represented using both graphical and textual notation, and is used to document design. I will not really be able to give you detailed test case design for UML diagrams, but I will tell you what sort of diagrams basically correspond to finite state automata and how the coverage criteria that we saw till now can be used on them. At the end of this lecture we will revisit the entire graph based testing that we have looked at in the past three weeks, and I will briefly recap all that we did.

(Refer Slide Time: 01:59)



The slide is titled "Finite state machines" in a blue header. It contains three bullet points: "A Finite State Machine (FSM) is a graph that describes how software variables are modified during execution.", "Nodes: States, representing sets of values for key variables (design variables).", and "Edges: Transitions, which model possible changes from one state to another. Transitions have guards and/or actions associated with them." There is a small HPTEL logo in the bottom left and a video feed of a woman in the bottom right.

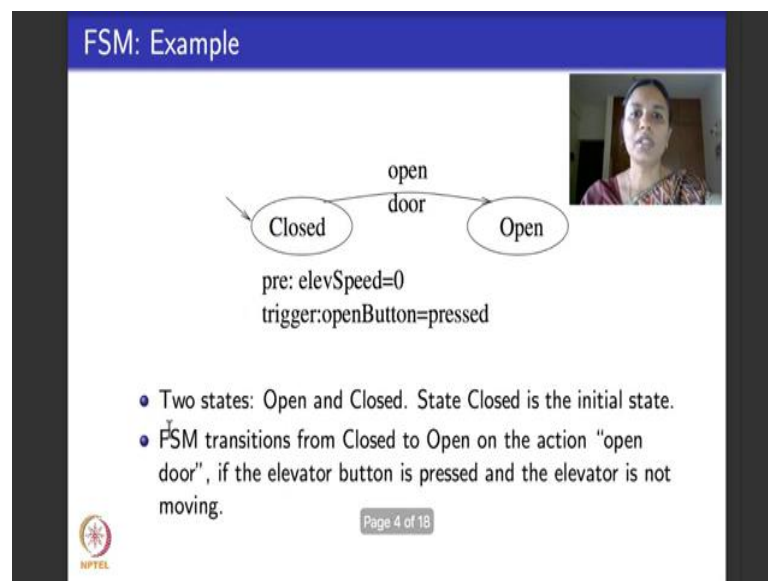
- A Finite State Machine (FSM) is a graph that describes how software variables are modified during execution.
- Nodes: States, representing sets of values for key variables (design variables).
- Edges: Transitions, which model possible changes from one state to another. Transitions have guards and/or actions associated with them.

What is the finite state machine? I hope most of you will be familiar with it. If you have done computer science or IT you would have seen a course called automata theory or formal languages or theory of computation; finite state automata are one of the first models that you will learn in the course. So, a finite state machine can be thought of as a finite graph, the nodes in the graph or what are called states, and the edges in the graph are what are called transitions. So, what do the nodes describe? The nodes typically describe what happens to certain variables and software at a certain point in time. They tell you the values that these variables take at that point in time. What are transitions describe? They describe how the machine or the program changes state from one stage to the other.

So, if a stage depicts the values of certain variables at a certain point in time, transition might mean executing a particular statement in the program, which results in change of

values of one or more states and this results in a new state. So, those are the edges of a finite state machine. Typically many interesting finite state machine models that I use for software design always have guards, conditions associated with edges, triggers associated with nodes and so on. So, here is a simple example of a finite state machine, it has 2 states, 2 nodes; one called closed, one called open. States are always given these names with special identifiers which help you to identify what these states are while modeling using finite state machines.

(Refer Slide Time: 03:23)

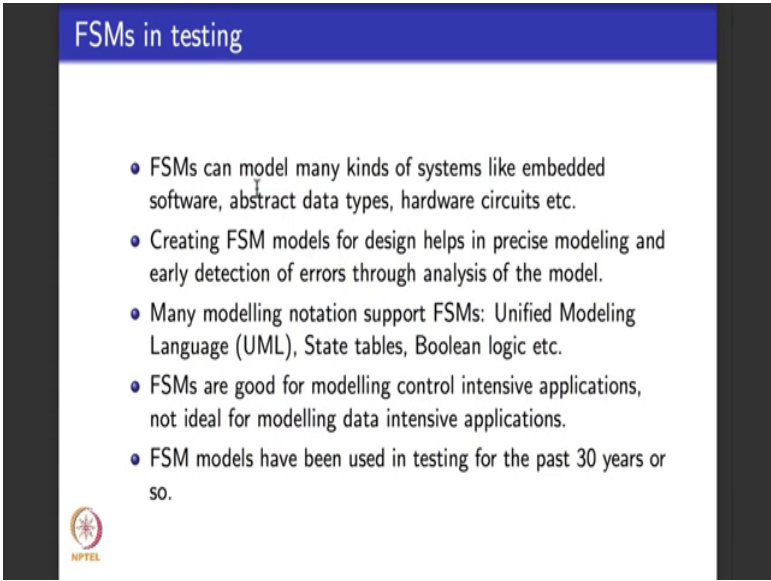


As we have seen till now there is an incoming arrow at the state closed. So, this says that closed is an initial state. This small, toy finite state machine can be thought of as the finite state machine corresponding to an elevator. It is a very highly abstract machine, but it represents an elevator. The elevator could be, door could be the closed or it is open and it says that there is a transition or an edge from the state closed to the state open, when somebody presses this open door button.

You could think of it as there being a door button and then when you press that open door button, the state machine changes from state closed to state open. And what are the preconditions for this transition or this state change to happen? Preconditions, here it is abbreviated as pre in short for precondition, it says the elevator speed should be 0, which basically means that the elevator should not be moving, if it moves the door cannot be opened and it says that somebody should have pressed the open door button.

So, the precondition says that elevator should not be moving, its speed be 0, and the trigger is the open button command should have been pressed by somebody. If this happens and the machine transitions from closed to open. Of course, this is just a small machine, it does not really describe all the transitions or the full behavior or design of an elevator, I just took this abstract level description to explain what a finite state machine is to you.

(Refer Slide Time: 05:17)



The slide has a blue header with the text "FSMs in testing". Below the header is a white rectangular area containing a bulleted list of five points. At the bottom left of this white area is a small circular logo with a star and the text "NPTEL" below it.

- FSMs can model many kinds of systems like embedded software, abstract data types, hardware circuits etc.
- Creating FSM models for design helps in precise modeling and early detection of errors through analysis of the model.
- Many modelling notation support FSMs: Unified Modeling Language (UML), State tables, Boolean logic etc.
- FSMs are good for modelling control intensive applications, not ideal for modelling data intensive applications.
- FSM models have been used in testing for the past 30 years or so.

So, how are finite state machine useful for us in testing? Finite automata are very classical models, any course and automata theory or theory of computation would begin with them they are very robust models, but we would not really look at finite state automata and their theory and what they mean because our focus is to be able to purely understand it from the point of view of testing. Surprisingly, finite state machines have been around in testing for more than 30 years now, and they have always been considered is popular models that have been used for designing test cases to achieve some goal or the other with reference to testing.

So, typically finite state machines, what do they model? The model design of embedded software; software that sits in cars, planes, phones, toys etcetera. They model several abstract data types like for example, in the last lecture we saw queue with an abstract data type, right. So, finite state machines can model such abstract data types, almost all

of hardware, hardware circuits can be modeled using finite state machine. Several protocols can be modeled using finite state machine.

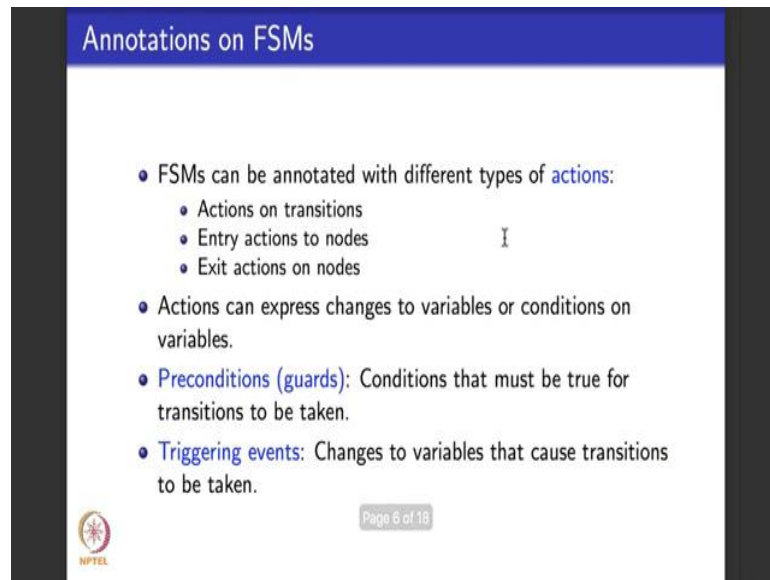
So, a large class of design and design element can be modeled using finite state machines and why does this help? Typically finite state machines model design and when I create FSM models for design, I am doing it before I write code. In fact, if my model is good enough then there are several tools that process finite state machine design models and automatically generate code.

So, when I come up with formal models of design, it is been found by past experiences in several projects, that such models help you to identify, test and detect errors early even before you begin coding, so that these did errors if detected early can save development cost and time. And the other thing is many as I told you many popular modeling notations also support finite state machines. The most popular modeling notation is that of UML. UML has a diagram one of it is diagrams is that of finite state machines, the other diagram that it has a state charts which can be thought of as finite state machines with hierarchy and concurrency specific finite state machines.

And the other thing to note is that whenever I have a control intensive applications right like elevator, something controlling something a software controlling an entity right controlling a piece of hardware or a system, finite state machines are considered to be very good models for that. But suppose I have a data intensive application like something that handles a database server, something that processes data and renders it as high speed GUI, FSMs are not considered good models for such things, we typically do not work with finite state machines for them right.

So, FSM, the summary of this slide is that finite state machines are reasonably popular. Fairly large class of software can be modeled the design of such software can be modeled using finite state machines.

(Refer Slide Time: 08:24)



The slide is titled "Annotations on FSMs" in a blue header. It contains a bulleted list of annotations for Finite State Machines. The first bullet point states that FSMs can be annotated with different types of actions, followed by a sub-list: actions on transitions, entry actions to nodes, and exit actions on nodes. The second bullet point explains that actions can express changes to variables or conditions on variables. The third bullet point defines "Preconditions (guards)" as conditions that must be true for transitions to be taken. The fourth bullet point defines "Triggering events" as changes to variables that cause transitions to be taken. In the bottom left corner is the NPTEL logo, and in the bottom right corner is a small box indicating "Page 6 of 18".

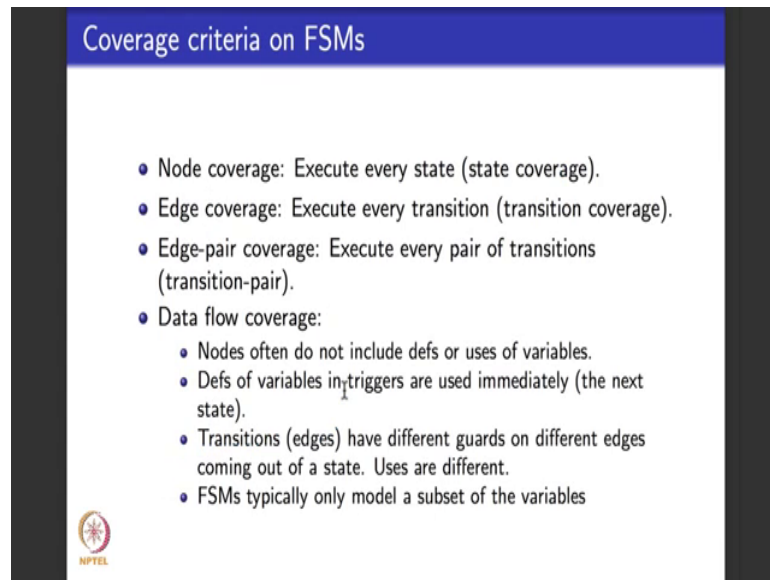
- FSMs can be annotated with different types of **actions**:
 - Actions on transitions
 - Entry actions to nodes
 - Exit actions on nodes
- Actions can express changes to variables or conditions on variables.
- **Preconditions (guards)**: Conditions that must be true for transitions to be taken.
- **Triggering events**: Changes to variables that cause transitions to be taken.

Page 6 of 18

So, it helps to look at our graph coverage criteria on finite state machines and see what they mean. Before we move on I would like to mention to you that finite state machines are beyond just simple graphs. When we saw structural coverage criteria we saw plane graphs and when we saw data flow coverage criteria we saw graphs that were annotated with definitions and uses, finite state machines have lot more annotations than data flow graphs.


They can be annotated with different types of actions, actions could be on the transitions of the machine, actions could be on the node specifically as actions that dictate when to enter a node, actions could be on the node dictating when to exit from the node; and what do these actions represent? They represent change of values of variables, they represent change of evaluation of conditions, several different things right. As we saw in the elevator, a small example, there could be preconditions associated with nodes that tell you when to take a transition out of a node. There could be conditions or guards associated with edges that tell you when to take a particular edge, they could be triggering events and so on. So, finite state machines are graphs, but they have a lot of extra add on information to them.

(Refer Slide Time: 09:41)



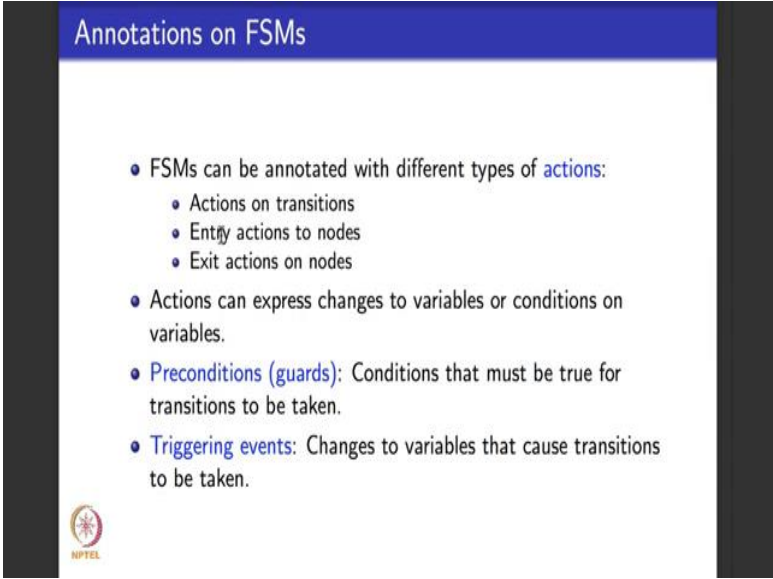
Coverage criteria on FSMs

- Node coverage: Execute every state (state coverage).
- Edge coverage: Execute every transition (transition coverage).
- Edge-pair coverage: Execute every pair of transitions (transition-pair).
- Data flow coverage:
 - Nodes often do not include defs or uses of variables.
 - Defs of variables in triggers are used immediately (the next state).
 - Transitions (edges) have different guards on different edges coming out of a state. Uses are different.
 - FSMs typically only model a subset of the variables




Now, let us look at the structural coverage criteria that we saw and see what they mean. So, simplest structural coverage criteria that we saw was node coverage. In the context of finite state machine it means execute every state. That makes a lot of sense no, when it comes to saying that if a machine models are designed, then you execute every state in that design right. The next coverage criteria is edge coverage, it means execute every transition which means execute every state change. The third coverage criteria is edge pair coverage, execute every pair of transitions. Now path coverage, prime path coverage is not very useful, because we do not really look at loops as they come from control flow graphs in finite state machines, what would be useful is specified path coverage. As we go down the course we will see some more finite state machines where this could be useful for us.

(Refer Slide Time: 10:49)



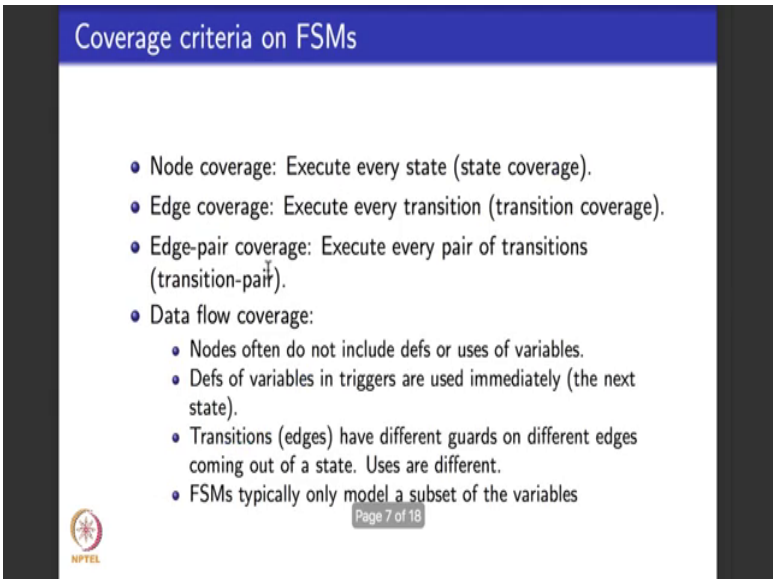
Annotations on FSMs

- FSMs can be annotated with different types of **actions**:
 - Actions on transitions
 - Entry actions to nodes
 - Exit actions on nodes
- Actions can express changes to variables or conditions on variables.
- **Preconditions (guards)**: Conditions that must be true for transitions to be taken.
- **Triggering events**: Changes to variables that cause transitions to be taken.




Now, let us move on and look at data and flow coverage. In the data flow coverage when it comes to applying for finite state machines definitions and uses can get very complicated. If you go back to the previous slide I told you that there could be actions associated with edges or transitions and then there could be actions associated with nodes and those actions further on could be entry actions or exit actions.

(Refer Slide Time: 11:05)



Coverage criteria on FSMs

- Node coverage: Execute every state (state coverage).
- Edge coverage: Execute every transition (transition coverage).
- Edge-pair coverage: Execute every pair of transitions (transition-pair).
- Data flow coverage:
 - Nodes often do not include defs or uses of variables.
 - Defs of variables in triggers are used immediately (the next state).
 - Transitions (edges) have different guards on different edges coming out of a state. Uses are different.
 - FSMs typically only model a subset of the variables



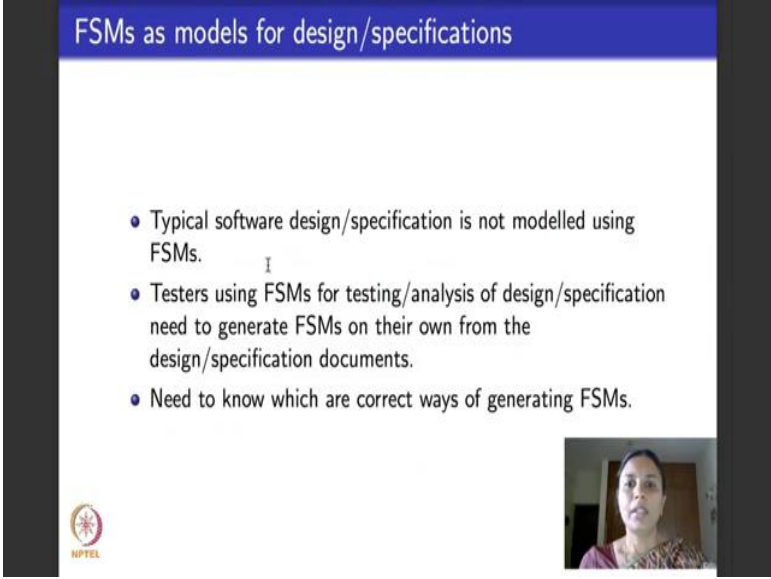
Page 7 of 18

So, there could be several different kinds of definitions associated with nodes and edges, and the other thing to be noted is that in when you look at control flow graph of

code when we say annotations and edges uses on edges, they typically occur as predicates that correspond to branching, one will be one predicate the other one will be the negation of that predicate. In finite state machine, 2 edges coming out of the same node could have very different kinds of guards that are labeling them. Some of them could use a certain variable, some of them need not use the same variables at all they could use a completely different set of variables.

So, the users on each edge that comes out of a node could be very different from the other edge that comes out of a node. So, data flow coverage criteria gets a little cumbersome when we have to work with the term finite state machine. Instead what we will do is next week I will introduced you to logical coverage criteria; logic coverage criteria come in handy when it comes to handling data with FSM, we will revisit finite state machines at that time right.

(Refer Slide Time: 12:07)



The slide has a blue header with the text "FSMs as models for design/specifications". Below the header, there are three bullet points:

- Typical software design/specification is not modelled using FSMs.
- Testers using FSMs for testing/analysis of design/specification need to generate FSMs on their own from the design/specification documents.
- Need to know which are correct ways of generating FSMs.

In the bottom right corner of the slide, there is a small video inset showing a woman speaking. In the bottom left corner, there is a small logo with the text "NPTEL" below it.

So, the other thing I told you is that finite state machines typically model software design or specification. And when you look at the large software organization none of the designers or architects typically sit down and draw or document the finite state machine for you. They typically write design or specifications as English statements and sometimes maybe some sketches and other diagrams, and whenever they are missing and if you find them to be useful for testing, as a tester the owners is on you to be able to

draw finite state machines. So, it helps to get some idea of what exactly are finite state machines and what do they represent when it comes to modeling design of software.

(Refer Slide Time: 12:50)

Generating FSMs

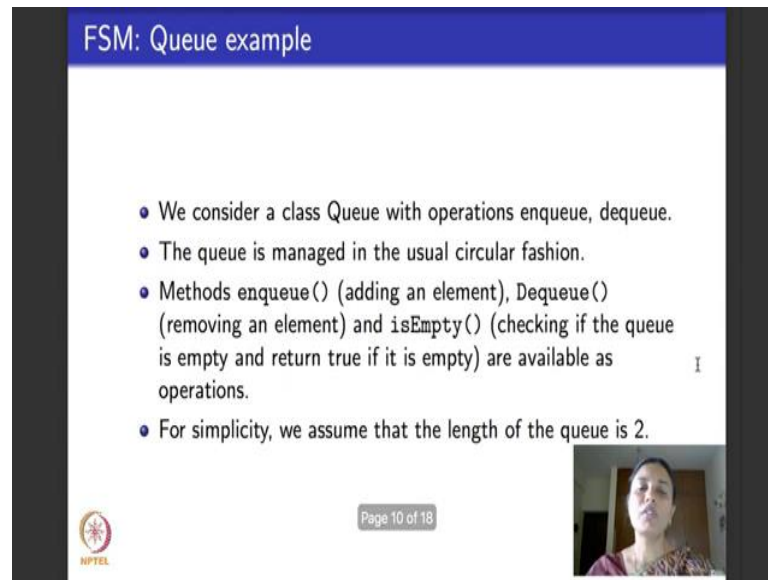
- Control flow graphs are not FSMs representing software/code.
- Call graphs are also not FSMs representing software/code.
- We need to consider values of variables to represent states of FSMs and statements/actions that result in change of values of variables (states) result in transitions.

Page 9 of 18

So, one thing to note is that control flow graph which we learned how to draw, that is not a finite state machine corresponding to any design or specification or even code. It is not a finite state machine because control flow graph does not give you values associated with variables in states. So, it can never really be a finite state machine in the sense that we introduced it to be. Similarly call graphs which was another model that we saw when we saw design elements which talked about how one module calls another module, the call graphs are also not really finite state machines, again because they do not debit data associated with states and the finite state machines.

What we need to do when we define finite state machines is that we need to consider values of variables that are present in the code. Each state of a finite state machine can be thought of as a tuple containing the values of the designated set of variables at any specified point in time and then transitions in the finite state machine tell you how when some statement in the program executes, the values of one or more variable changes and how it affect these transitions.

(Refer Slide Time: 14:02)



FSM: Queue example

- We consider a class Queue with operations enqueue, dequeue.
- The queue is managed in the usual circular fashion.
- Methods enqueue() (adding an element), Dequeue() (removing an element) and isEmpty() (checking if the queue is empty and return true if it is empty) are available as operations.
- For simplicity, we assume that the length of the queue is 2.

Page 10 of 18

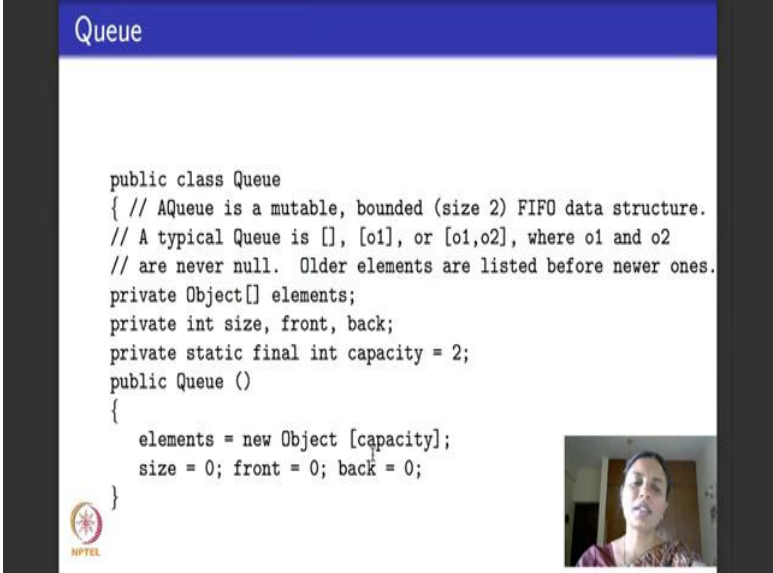
So, what we will do is that if you remember in the last lecture I had introduced you to this example queue example and we looked at one sequence in constraint in the queue.

So, the class queue typically comes with several methods that support operations on queues, one method is that enqueue where you add an element of the queue, another method is dequeue where you remove an element from the queue, and you could also do things like methods which query and tell you is the queue empty is the queue full and so on. So, and we saw that a simple sequencing constraint like, for dequeue to happen and enqueue should have happened in the past could be described and tested, but more complex sequencing constraints saying that the number of enqueues at any point in time should be greater than or equal to the number of dequeues cannot be tested by directly writing them as sequencing constraints. We said last time that we would need finite state machines for testing that.

So, what I will do today is we will look at this queue example, and I will give you an idea about how a finite state machine for such a queue abstract data type will look like and because see when I talk about a queue, I need to implicitly assume a bound on the length of the queue. Suppose I say I can keep adding to the queue then in some sense the state space of the finite state machine will become infinite and it will no longer be finite state. So queues, we assume or of some bounded length for the sake of illustration to make things simple I have assumed that the length of the queue is just two. Of course, it

does not correspond to a realistic model, to make it realistic you could assume that the queue is of fixed length n for some arbitrarily large n that you wanted to be, the same reasoning will extend for that length also.

(Refer Slide Time: 15:49)



```
public class Queue
{ // AQueue is a mutable, bounded (size 2) FIFO data structure.
  // A typical Queue is [], [o1], or [o1,o2], where o1 and o2
  // are never null. Older elements are listed before newer ones.
  private Object[] elements;
  private int size, front, back;
  private static final int capacity = 2;
  public Queue ()
  {
    elements = new Object [capacity];
    size = 0; front = 0; back = 0;
  }
}
```

So, here is a part of the code for the class queue. So, what is a queue? Queue is a mutable, bounded first in first out data structure. I have assumed the bound to be 2 for the sake of simplicity. So, what could queue look like? It could be empty it could just have one object or it could have 2 objects, we assume that these objects are not null and because it is first in first out the older elements are listed before the elements that are new in the queue. So, queue has things like size front and back, and it also its main state element is set of an elements. We assume that the capacity of the side queue is 2. Initially, its size is 0, there is nothing in the front nothing in the back and element is something that I want to add now.


(Refer Slide Time: 16:37)

Queue, contd.

```
public void enqueue (Object o)
throws NullPointerException, IllegalStateException
{ // Effects: If argument is null throw NullPointerException
// else if queue is full, throw IllegalStateException,
// else make o the newest element of this
if (o == null)
throw new NullPointerException ("Queue.enqueue");
else if (size == capacity)
throw new IllegalStateException ("Queue.enqueue");
else {
    size++;
    elements[back] = o;
    back = (back+1) % capacity;
}
```

NPTEL

Page 12 of 18



So, there are 2 methods that I have, in this slide I have presented enqueue method.


(Refer Slide Time: 16:39)

Queue, contd.

```
public Object dequeue () throws IllegalStateException
{ // Effects: If queue is empty, throw IllegalStateException,
// else remove and return oldest element of this
if (size == 0)
throw new IllegalStateException ("Queue.dequeue");
else {
    size--;
    Object o = elements [(front % capacity)];
    elements[front] = null;
    front = (front+1) % capacity;
    return o; }
}
```

NPTEL

Page 13 of 18



In this slide I have presented dequeue method of course, the queue class can have lot more methods like I told you, you could have a method just check if the queue is empty, you could have a method that checks if the queue is fully and so on. I have not described all those methods. For simplicity I have just given you enqueue and dequeue methods. So, we will see what enqueue does. So, enqueue inserts an object o into the queue and whenever it is not possible to insert, it could throw exceptions. It throws a null pointer

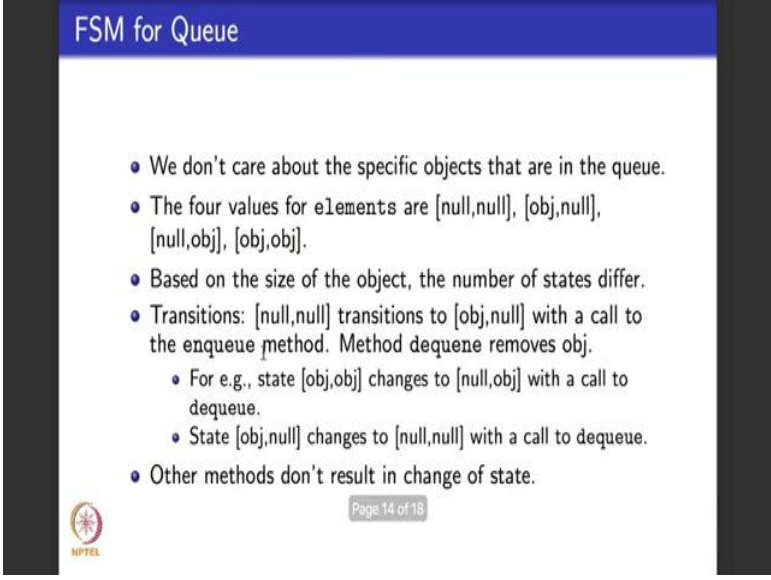
exception if the argument to be inserted is null. We assume that the objects to be inserted or non-null as I told you here we assume that o 1 and o 2 are never null.

And it throws an illegal state expression if the queue is already full. So, it cannot insert any more element into the queue. So, this is the code for the method, it says if the object to be inserted is null you throw a null pointer exception or if the size is same as the capacity of the queue then you throw a illegal state exception, otherwise you increase the size and add the object over to the elements and increase the capacity, is this clear.

What is dequeue do? Dequeue tries to remove the topmost element from the queue. If the queue is empty and there is nothing to remove it will throw an exception, illegal state exception, otherwise it will remove the oldest element or the topmost element from the queue. So, if the size of the queue is 0, then you throw illegal state exception. Otherwise you decrease the size, you remove that object in the front because it is queue which FIFO first and first out and then you reset the new thing and adjust the capacity accordingly, and then the object that you removed you return that object.

So, now suppose we have to model the queue, the contents of the queue as a finite state machine, what would be a correct approach? The correct approach would be to first think about what is the state of the queue at any point in time. Remember that states in a finite state machine talk about the values of all the variables involved in a program or the design of a program and it actually depicts the existence, the state of the system as it would exist in real life. So, when I look at the queue data structure what is it is state? It is state is the content of the queue, what is the contents of the queue at any point in time.

(Refer Slide Time: 19:08)



The slide is titled "FSM for Queue" in a blue header. It contains a list of bullet points explaining the states and transitions of a queue. At the bottom left is a small circular logo with the text "NPTEL" below it. At the bottom center is a small rectangular box with the text "Page 14 of 18".

- We don't care about the specific objects that are in the queue.
- The four values for elements are [null,null], [obj,null], [null,obj], [obj,obj].
- Based on the size of the object, the number of states differ.
- Transitions: [null,null] transitions to [obj,null] with a call to the enqueue method. Method dequeue removes obj.
 - For e.g., state [obj,obj] changes to [null,obj] with a call to dequeue.
 - State [obj,null] changes to [null,null] with a call to dequeue.
- Other methods don't result in change of state.

So, we do not really care about the specific objects in the queue, all that we want to know is that the; what are what is there in the queue. The queue could be have size at most 2. So, the queue could be empty there could be one non null object at the beginning of the queue, front of the queue, there could be one non null object at the back of the queue or the queue could be full, two non null objects right. So, these could be the 4 values of this variable element which describe the contents of the queue.

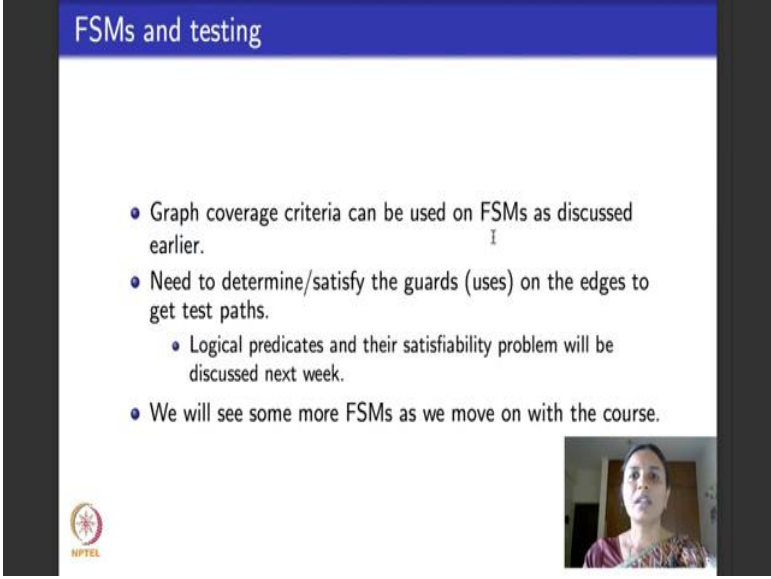
You go back to the queue code queue consists of elements and the queue is of size 2. So, all that I am tracking is what is the current content of the queue. Is the queue empty, does the you have one non null object, does the queue have one non null object in the front or at the back, does the queue have 2 non null objects? So, those are these values and then these are the states right. So, there could be it several different states you can compute how many states would be there, but about 6 of them would be reachable states. What would be the transitions. Like for example, suppose I have something like this [object, object] which what are the methods of the method calls that will change the state of the queue will a method call which says is the queue empty change the state? No, right?

Because what will is the queue empty do? It will basically look at the queue check whether it is empty or not if it is empty it will say true if it is non-empty it will return false. It is not going to be able to change the content of the queue. But methods like enqueue and dequeue will change the state of the queue because they alter the variable

elements they alter this variable elements. When a enqueue, I add something it will the element when I dequeue I remove something from the elements. Like for example, here is the transition suppose I begin with [null null], that changes to object null when I enqueue an object and similarly when I dequeue an object, if I have a full queue and I dequeue the topmost object, that gets replaced with empty and then it becomes like this. If the queue is a capacity one which it looks like this it has 1 non null object if I dequeue again, then the queue changes to a null queue right.

So, this is how the state machine for a queue would look like. I have not really drawn the state machine pictorially I would like to leave that as a small exercise for you to do. Assume that these are the states of a state machine and draw edges to depict the change of states, and label those edges with method names like enqueue and dequeue which tell you when you do an enqueue and when you do a dequeue, how the state changes in terms of elements getting added or removed from the queue respectively? Try and draw this finite state machine as a small exercise that you could do for yourself.

(Refer Slide Time: 22:05)



The slide is titled "FSMs and testing" in a blue header. It contains a list of four bullet points:

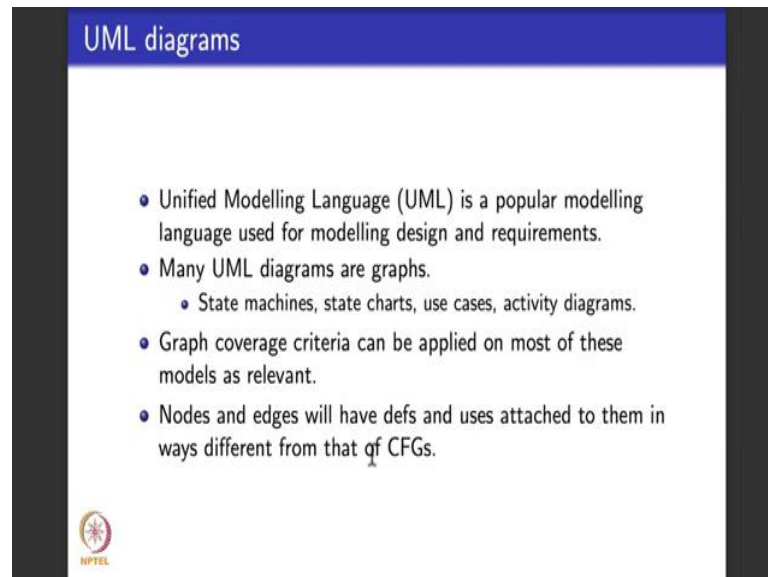
- Graph coverage criteria can be used on FSMs as discussed earlier.
- Need to determine/satisfy the guards (uses) on the edges to get test paths.
 - Logical predicates and their satisfiability problem will be discussed next week.
- We will see some more FSMs as we move on with the course.

In the bottom right corner, there is a small video inset showing a woman speaking. In the bottom left corner, there is a small logo for "HPPTEL".

Now, when it comes to testing, as I told you any kind of graph coverage criteria that deals with structural coverage criteria can be used except for prime paths,. We specifically use node coverage, edge coverage, edge pair coverage and specified path coverage when it comes to finite state machines. Coming up with test paths for these coverage criteria can be non trivial because you have to solve for the actions and the

guards that come in state machines. We will deal with it when we look at how to solve logical predicates next week, we will also see some more examples of concrete finite state machine as we move on in the course.

(Refer Slide Time: 22:56)

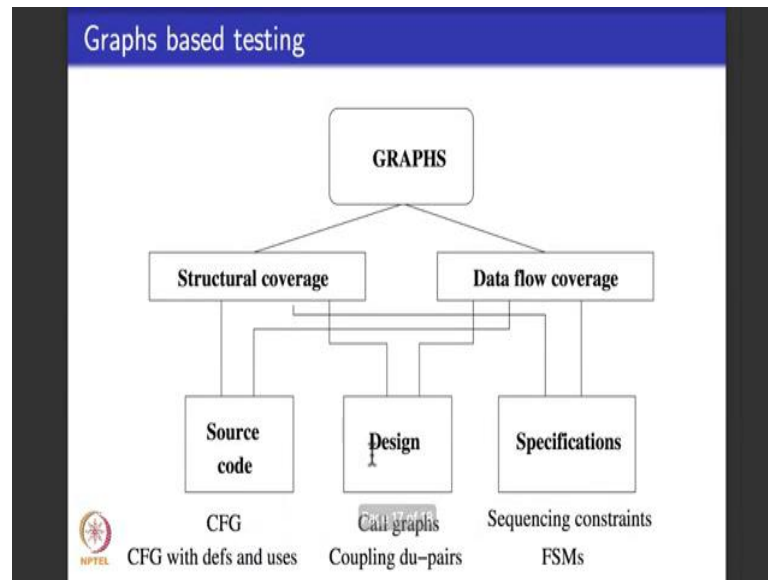


So, that is all I wanted to talk to you about finite state machines. I will briefly spend some time looking at some part of UML diagrams and tell you that the coverage criteria that we have seen till now can also be used to test UML diagrams. UML, in short for Unified Modeling Language, is a very popular modeling language used for modeling designs of systems.

There are about 14 or 17 UML diagrams, and here are the ones that look a lot like graphs: state machines, state charts, activity diagrams. They are basically some special kinds of graphs and the kind of coverage criteria that we have seen specifically path coverage criteria, specified path coverage criteria, can be used to test most of these graphs. One is to thing to be noted that each of this graph is very different from a typical control flow graph so you have to be careful when we define test cases and path coverage criteria on them.

As I told you I really would not be able to do UML now for most part of the course. If time permits towards the end of the course maybe we could look at UML diagrams and see is testing specific to UML diagrams.

(Refer Slide Time: 24:00)



We have come to an end of graph based testing. Next week onwards, I will begin logic predicate based testing. Here is a quick recap of what we have seen till now. The primary structure or model that we worked with for the past three weeks was that of graphs. The 2 main kinds of coverage criteria that we saw on graphs were structural coverage criteria and data flow coverage criteria. We defined these coverage criteria purely graph theoretically, we saw what are the various structural coverage criteria. If you want to list them, we saw node coverage, edge coverage, edge pair coverage, complete path coverage, specified path coverage, prime path coverage.

Then we saw subsumption, how each coverage criteria subsumes one or more of the other. Then we augmented graphs with data specifically with data definitions and data uses. Then we saw what are called def use paths or du-paths, and then we saw three main data flow coverage criteria: all defs coverage, all uses coverage and all du-paths coverage. Then what we did where we applied this graph coverage criteria that we learnt to source code, to design and to specification. In source code, we specifically learned how to draw a control flow graph for various code snippets, applied them to a full example and so, how the various structural coverage criteria can be used to test CFGs.

Then we augmented CFGs with defs and uses and saw how to use data flow coverage criteria on this augmented CFG. Then we moved on to design I gave you the basics of designs integration testing, then we saw call graphs and we saw how to apply structural

coverage criteria on call graphs; then we saw coupling du-pairs: variables that are defined in one used in the other, right from actual parameter to formal parameter, from a caller method to a callee method, and augmented this data flow coverage criteria to coupling variables. Finally, the last couple of lectures we saw how to use graph coverage criteria on specifications. I told you how to use graph coverage criteria to model and test sequencing constraints and in today's lecture we saw finite state machines. This week's assignment will deal with this part, I will give you assignments the talk about data flow coverage criteria design and specifications.

Next week I will upload a video on how to solve that assignment. Please try to do it before you see the video once, it give you a good practice, and that will be the end of graph coverage criteria for this part of the course. We will move on to looking at logical predicates, define what are the coverage criteria on logical predicates and how to apply them to code and to specifications.

Thank you.

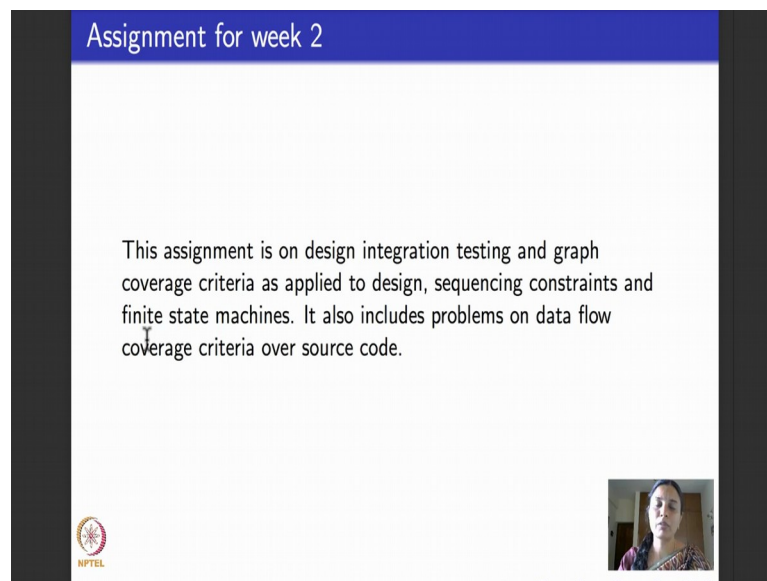
Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 20

Assignment 4: Graph coverage criteria: Data flow coverage over source code, design and specifications

Hello everyone. Welcome to week 5: the goal is to finally leave graph coverage criteria and move on to other data structures and models. But before we do that the first video for week 5 would be, telling you how to solve the assignment that was uploaded for week 4 because there was a programming question and you were asked to answer some questions about def-use paths on that program. So, I would like to spend this first video for week 5 trying to see whether you have solved it and to help you, if you have not been able to solve and you could also use this video to be able to crosscheck and see if your answers were right or wrong.

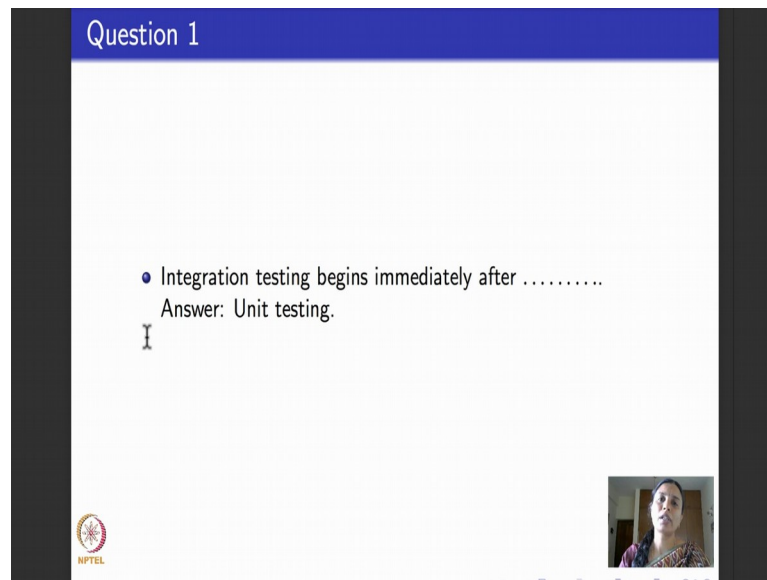
(Refer Slide Time: 00:55)



The screenshot shows a presentation slide with a blue header bar containing the text "Assignment for week 2". The main content area is white and contains the following text: "This assignment is on design integration testing and graph coverage criteria as applied to design, sequencing constraints and finite state machines. It also includes problems on data flow coverage criteria over source code." In the bottom right corner, there is a small video inset showing a woman speaking. In the bottom left corner, there is a small circular logo with the text "IITEL" below it.

What was the assignment? The assignment for week 4 was on design integration testing, graph coverage criteria as it was applied to design, sequencing constraints and we also saw one problem on data flow coverage criteria.

(Refer Slide Time: 01:08)



Question 1

- Integration testing begins immediately after

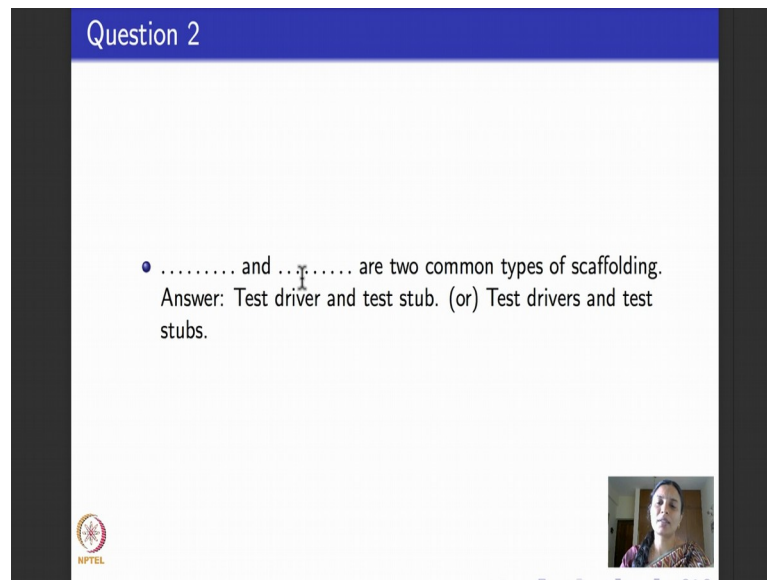
Answer: Unit testing.

NPTEL

So, what I will do is, I will walk you through one question after the other in the assignment and tell you what the correct answer is and what it means, what does the correct answer mean. So, the first question in that assignment was the fill in the blanks question, it asked which is the phase or stage at which integration testing sits?

That means which is the phase immediately after which integration testing comes. If you remember the V model or the waterfall model that includes testing, Integration testing comes immediately after unit testing, when the individual unit tested modules are ready. I am ready to put them together and do design integration testing. So the answer to the first question would be unit testing.

(Refer Slide Time: 01:51)

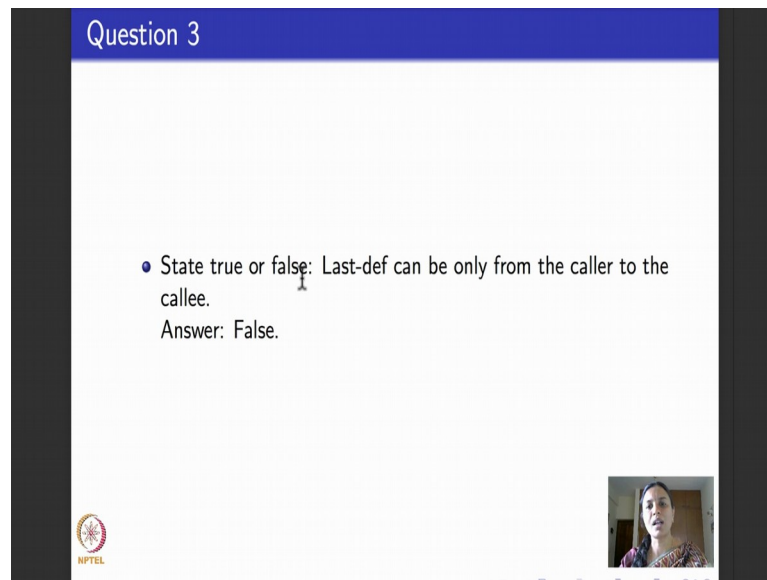


The screenshot shows a presentation slide with a blue header bar containing the text "Question 2". The main content area is white and contains a bulleted question: "..... and are two common types of scaffolding." followed by the answer: "Answer: Test driver and test stub. (or) Test drivers and test stubs." In the bottom right corner, there is a small video inset showing a woman with dark hair, wearing a patterned top, looking towards the camera. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.

Second question asks when I do integration testing, I do what is called scaffolding. If you remember the modules that we saw when we looked at classical design integration testing, it says, which are the two most common types of scaffolding? Two most common types of scaffolding are test drivers and test stubs. If you remember, when we do top down integration or bottom up integration, I may or may not have all the modules ready.

So, I write a test stub when I do not have a module ready, but I want it to behave as if a module is ready. So, it is like a dummy module and when do I do a test driver? When I want the dummy module to be able to call the lower level modules, when I do bottom up testing that is when I use test drivers. So, the correct answer to second question will be test stubs and test drivers.

(Refer Slide Time: 02:53)



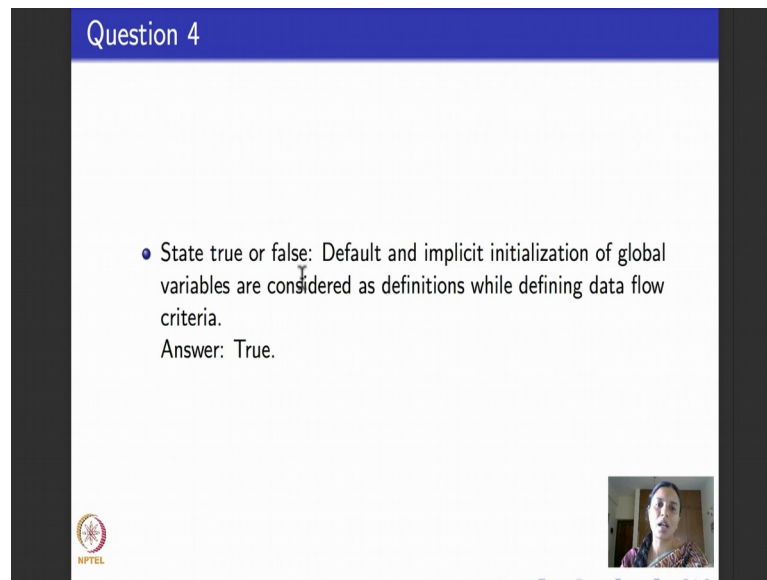
Question 3

- State true or false: Last-def can be only from the caller to the callee.

Answer: False.

The combinations that are given here basically let you answer them in any order. You should have got full marks if you answered them in any order or in singular or plural words. The third question asks; gives you a statement which is Last-def can only be from the caller back to the callee. From the caller to the callee, sorry not back to the callee. So, is it true or false? So it is obvious that it is false because even in the examples that we saw, if you just remember we saw quadratic root example, then we saw small toy examples or just the caller, callee definitions. Last-defs can also be from the callee back to the caller. So, they can be in two different ways, they can be from the caller method to the callee method and they can be given, returned back, values that are returned back from the callee method back to the caller method.

(Refer Slide Time: 03:44)



Question 4

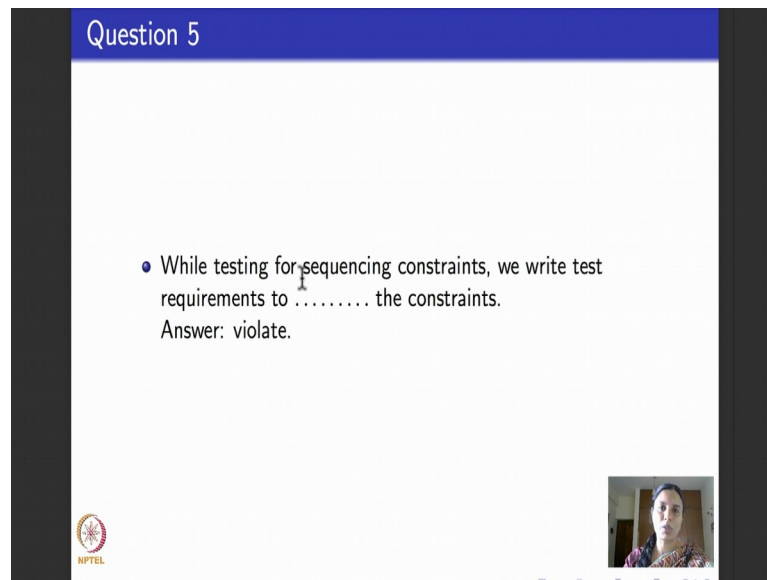
- State true or false: Default and implicit initialization of global variables are considered as definitions while defining data flow criteria.

Answer: True.

So, this statement is false; they can be both ways, it cannot be just one way. The fourth question asks again another true or false statement, it asks whether default and implicit initializations of global variables, specifically, when it comes to programs like C or Java; the programs that are written in C or Java, are they to be considered as definitions or not? Because they are implicit initializations, they are not explicitly given by the program, it does not mean that the values are not there.

So, they are initialized, the values are very much present. So they have to be considered as definitions even though they are not explicitly present as statements in the program. So, the answer to this question is true. Default and implicit initializations of global variables should be considered as definitions while working with data flow criteria.

(Refer Slide Time: 04:33)



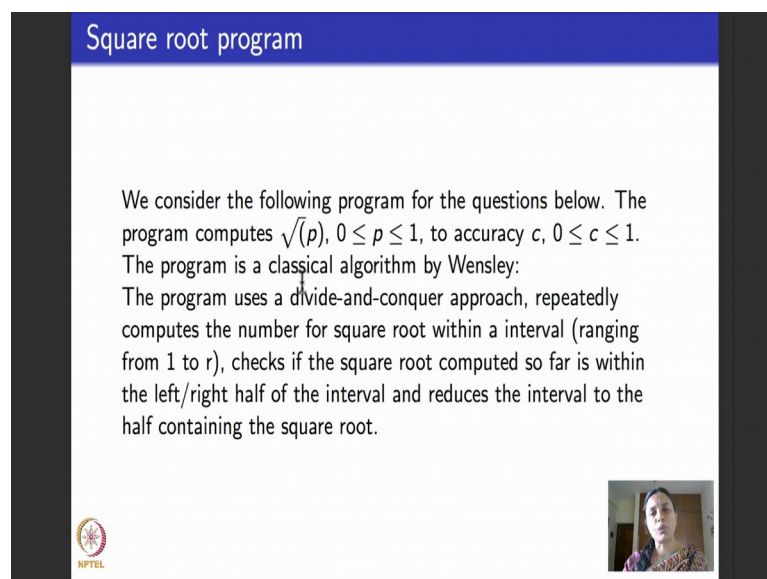
Question 5

- While testing for sequencing constraints, we write test requirements to the constraints.
Answer: violate.

The slide features a blue header with the text 'Question 5'. Below the header, a bullet point states: 'While testing for sequencing constraints, we write test requirements to the constraints.' The answer 'violate.' is provided below the bullet point. In the bottom right corner, there is a small video feed of a person. The NPTEL logo is visible in the bottom left corner.

The fifth question was the fill in the blank question. It asks you while you are testing for sequencing constraints, how do we write typically the test requirements for? If you remember we looked at this file ADT example and then that file ADT example had a few methods that it was exposing; open a file, read a file, write a file, close a file. So, typically we write test requirements to violate the sequencing constraint so that we can flag it as an error. So the answer to this fill in the blank question is, violate.

(Refer Slide Time: 05:12)



Square root program

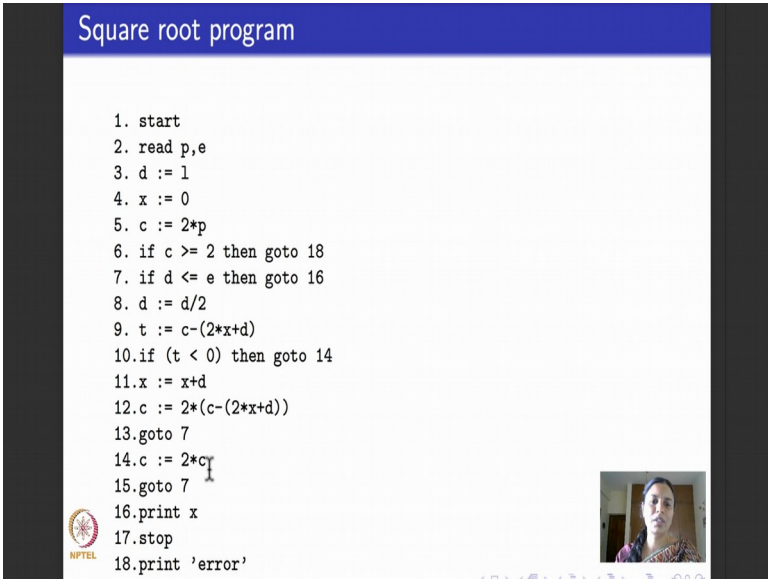
We consider the following program for the questions below. The program computes \sqrt{p} , $0 \leq p \leq 1$, to accuracy c , $0 \leq c \leq 1$. The program is a classical algorithm by Wensley: The program uses a divide-and-conquer approach, repeatedly computes the number for square root within a interval (ranging from 1 to r), checks if the square root computed so far is within the left/right half of the interval and reduces the interval to the half containing the square root.

The slide has a blue header with the title 'Square root program'. The main text describes a program for computing square roots using a divide-and-conquer approach. It mentions that the program computes \sqrt{p} for $0 \leq p \leq 1$ to an accuracy c where $0 \leq c \leq 1$. It attributes the algorithm to Wensley and describes its iterative nature, where it repeatedly computes the square root within an interval and narrows it down based on whether the current value is in the left or right half. A small video feed of a person is in the bottom right, and the NPTEL logo is in the bottom left.

From question 6 onwards; in the fourth assignment, you were given a program that was computing the square root. The program was given without any loops, but it had a loop. In fact it had more than one loop in the form of goto statements. I have taken this program from the classical papers by Sandra Rapps and Elaine Weyuker, 1982 paper, which I gave you as a reference for data flow testing. This program is from that paper, there were a few questions about that program, specifically related to dataflow coverage criteria.

So, what is the program doing? The program was trying to compute square root of p , where p was this number up to some kind of allowed accuracy and it is a classical algorithm by this Wensley. You can Google for it and check, it uses what is called a divide and conquer approach; it tries to look at first look at the interval 0 to p and try to see where the square root of p could be. Will it be in the first half or will it be in the second half, and based on what it says, it breaks the interval in two halves, repeats this procedure again. Breaks the next interval into two halves, repeats this procedure again and it narrows down the interval where square root of p comes and when it finally, terminates and outputs, it outputs a value that is almost the square root of p , modulo a small difference in accuracy.

(Refer Slide Time: 06:34)



The slide is titled "Square root program" in a blue header. Below the header, the code is written in a monospaced font, using a sequence of numbers and goto statements to represent a loop. The code is as follows:

```
1. start
2. read p,e
3. d := 1
4. x := 0
5. c := 2*p
6. if c >= 2 then goto 18
7. if d <= e then goto 16
8. d := d/2
9. t := c-(2*x+d)
10. if (t < 0) then goto 14
11. x := x+d
12. c := 2*(c-(2*x+d))
13. goto 7
14. c := 2*c
15. goto 7
16. print x
17. stop
18. print 'error'
```

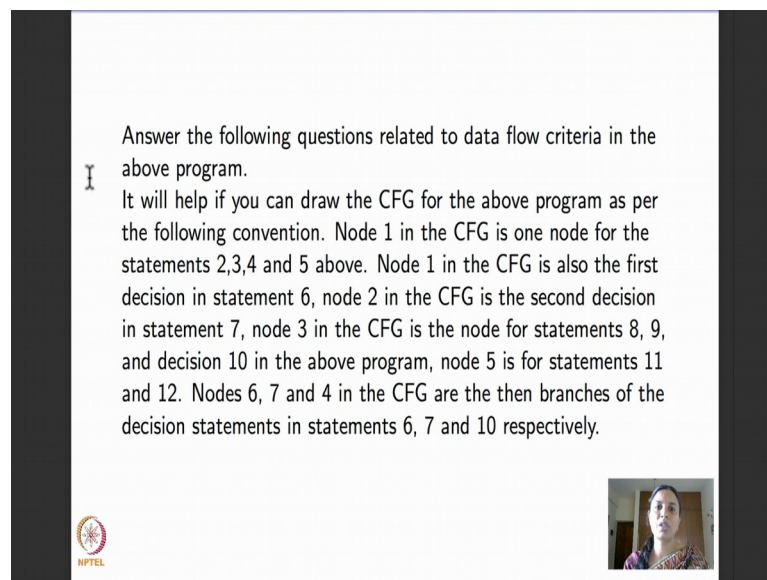
In the bottom right corner of the slide, there is a small video inset showing a woman with dark hair, wearing a patterned top, looking towards the camera.

So, here is the code of the program written specifically in long way that helps you to learn also how to draw control flow graphs. As I told you, the code, the program has

while loops not as while loops or for loops, it has them as goto statements. So, here I begin; I read the numbers, I do a set of initializations and then I say if this number is greater or equal to 2; then you go to 18, 18 means I cannot find square root; print error and it says if the number is less than equal to e; go to 16; 16 says I have found the square root print it.

So, then where is the loop? The loop is around here, so there is a if statement here; it computes d and t. If t is less than 0, it says go to 14; 14 does $2 \star c$ and then it says go back to 7. So, you do this then you could either go to 16 or you continue 8, 9, 10, 14 and go back to 7, so there is a loop here.

(Refer Slide Time: 07:35)



Answer the following questions related to data flow criteria in the above program.

It will help if you can draw the CFG for the above program as per the following convention. Node 1 in the CFG is one node for the statements 2,3,4 and 5 above. Node 1 in the CFG is also the first decision in statement 6, node 2 in the CFG is the second decision in statement 7, node 3 in the CFG is the node for statements 8, 9, and decision 10 in the above program, node 5 is for statements 11 and 12. Nodes 6, 7 and 4 in the CFG are the then branches of the decision statements in statements 6, 7 and 10 respectively.

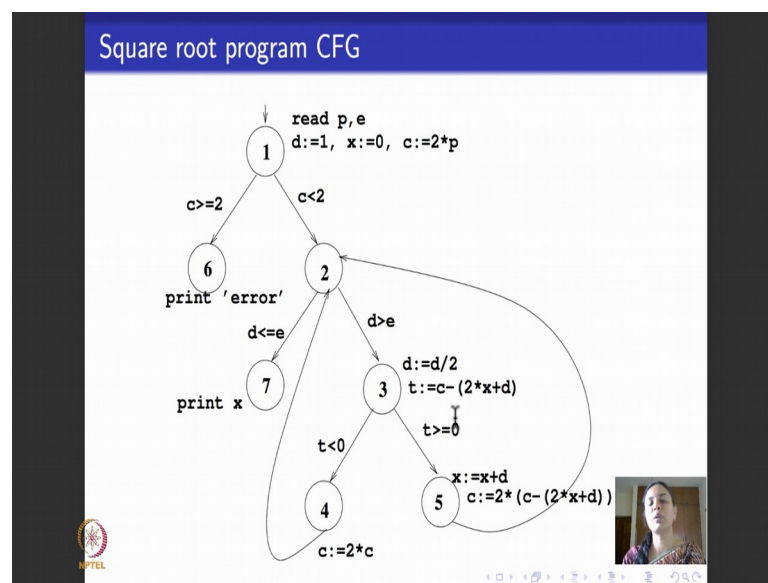
So, what was the question about? The question was about, first it asked you to draw a CFG. There were no marks allotted to drawing of the CFG. But the questions that were to follow were based on the CFG, especially four of those questions. The first one was not, the later questions were based on those of drawing the CFG and because this program has a lot of statements and the questions were based on parts in the CFG, I had also given you some guidelines on how to label the vertices of your CFG. So, these are the guidelines.

So, what it says is the first node in your CFG, call it node 1, should be one node that represents this bunch of statements that are in one after the other: 2, 3, 4 and 5. So, we will go back to the program statement 2; read p e, 3 this statement, d is equal to 1, 4, x is

equal to 0, 5 c is equal to 2 star p. All this at the beginning of the program will be node 1 in the CFG. That is what was given here and node 1 also includes this first decision statement at line number 6.

Then there is a branching into node 2 and some another node and in node 2, the second decision statement comes which means this decision statement is false. Second decision statement comes and then d by 2, t and so on; that is node 3 which is statements 8 and 9; node 3 in the CFG and so on.

(Refer Slide Time: 09:07)



So, here is how this CFG will look like, if you had made an attempt to draw it using pen and paper. So, node 1 represented with the initial node of the program, where the program computation begins. It represents these four statements read p, e, d is equal to 1, x assigned to 0, c is 2 star p, which are these four statements and it also represents the decision. If the if statement turns out to be true; that is if c is greater than or equal to 2, go to 18; 18 says print error. So in the CFG, you go to some node call it 6 as per the convention that I had given you to do in the question, which is labeled with the statement print error.

Otherwise you go to statement node 2 in the CFG, which is the next if statement; d is less than or equal to e. So, what you do here if d is less than or equal to e, then you print x go back to the code; if d is less than or equal to e, go to 16; 16 says print x. Otherwise d is greater than e, you come here, d is equal to d by 2; statements 8, 9, 10. So, d is equal to d

by 2 statements 8; t is equal to c minus 2 into x plus d , statement 9 and the decision at 10 is also modeled here.

Decision could be positive or negative, it checks if t is less than 0. If t is less than 0, it asks you to go to 14; where you do c is equal to $2 \star c$. If t is less than 0; then you do c is equal to $2 \star c$ and then what you do; you go back to 7; which is node 2 first. If t is greater or equal to 0; then you do two more statements and you go back to 7. So, these are the loops that I was telling you that the program has. Divides the number by 2, checks the range of the square root, takes the first half or the second half based on what the where the square root is, it keeps doing it till it reaches the value that is approximately the same as that of the square root. But these loops that you find in the CFG are not directly given as while statements or for statements in the program, they are given in terms of goto's.

So, I hope you all could get a CFG that looks somewhat like this. Even if it was not drawn like this, your CFG should have had 7 nodes and they should have been 3 decision points. The point 1 here, the point 2 here, point 3 here, corresponding to 3 if statements that were in the code; one here, one here and one here and these goto's tell you to loop back and that is what we have done here. The goto's loop back correctly.

Please do not get confused with the numbers 1, 2, 3, 4 that you find in the nodes here with the statement numbers in the program. These are different, they do not mean the same. So when I say goto 7 here, I mean go to statement number 7 in the program; statement number 7 in the program is actually node number 2 in the CFG. So, please do not mix up these two, so once you have drawn the CFG, you should be able to answer most of the questions that were asked.

(Refer Slide Time: 12:18)

Question 6(a)

- Does this program have an error? Manually investigate the program and say yes/no.

Answer: Yes. Statements 11 and 12 should be inter-changed.

I

NPTEL

So now we look at one question after the other and see what were the questions that were asked? So, the first question in fact, was not dependent on the CFG. It just asked you to look at the program, to study the program and find out on your own without resorting to any graph based testing or any other testing; does this program have an error? We typically give these kinds of questions in a testing course, to help you to appreciate that the goal of a tester is to be able to find an error.

So if you had done this, if you just studied the program and understood what it was doing with reference to divide and conquer algorithm, you will realize that the program did have an error. The error in the program was the order in which statements 11 and 12 were given to you, got interchanged. It should first compute this value, only then can it increment or reset the interval value of x . But the program was first incrementing x and then computing c , it will miss computing certain values of the square.



So, you take a small enough number and let the program go through it, you will realize that the statements 11 and 12 should be interchanged. So that is the error in the program, this program does have an error. So, the rest of the questions are aimed at helping you to understand data flow coverage and specifically can we use any of the data flow coverage criteria that we saw to find this error, the error that says statement 11 and 12 were indeed interchanged.

(Refer Slide Time: 13:46)

Question 6(b)

- Does the set of test paths $\{[1, 6], [1, 2, 3, 4, 2, 3, 5, 2, 7]\}$ satisfy edge coverage?

Answer: Yes.



So, the second question was does the set of test paths $[1, 6]$; $[1, 2, 3, 4, 2, 3, 5, 2, 7]$ satisfy edge coverage? So, please remember these test paths $[1, 6]$ and then $[1, 2, 3, 4, 2, 3, 5, 2, 7]$. We will go back to the CFG $1, 6$ and then $1, 2, 3, 4, 2, 3, 5$ and then 7 , does this satisfy edge coverage? Yes, it does because $1, 6$ covers this edge, then $1, 2, 3$; we have covered these three edges, 4 ; this edge is also done. $2, 3$ once again, but we have already covered it; does not matter; $5, 3, 5$; then $5, 2$ and $2, 7$. So, the answer to that question is yes, it does satisfy edge coverage. In fact it covers the edge $2, 3$ twice, but that does not matter, it visits every edge exactly once.



(Refer Slide Time: 14:49)

Question 6(c)

- Which of the following criteria does the set of paths $\{[1, 6], [1, 2, 3, 5, 2, 3, 5, 2, 7], [1, 2, 7], [1, 2, 3, 4, 2, 3, 5, 2, 7], [1, 2, 3, 4, 2, 3, 4, 2, 7], [1, 2, 3, 5, 2, 3, 4, 2, 7]\}$?

- All-defs coverage.
- All-uses coverage.
- All du-paths coverage.

Answer: 2nd option above.



The second question asks, gives you a set of test paths, lot of them in fact, and it asks you; lists three coverage criteria and asks you, which amongst these three coverage criteria are satisfied by this test path? So, if you see the test paths before looking at these test paths, let us go and look at the control flow graph. As I told you, control flow graph has how many branches? There is one branch here, there is one branch here and one branch here corresponding to the three if statements.

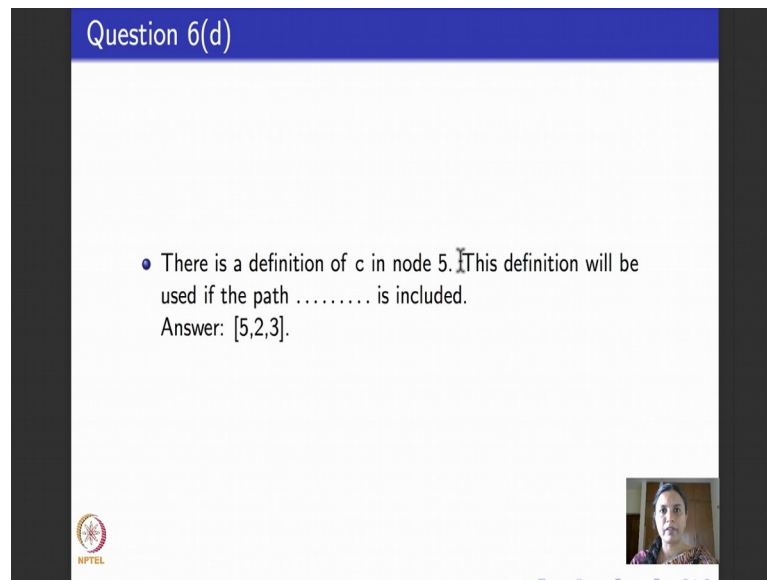
The third if statement had goto's, so it was like having a loop back. So, if I want to do edge coverage or any kind of coverage, I should cover this 1, 6 separately because it is a standalone path and then I can do all these. So, these paths that are given here--- do 1, 6 separately and then they had this one which we saw in the last question about edge coverage and then it had 1, 2, 7.

So, let us go back and see what 1, 2, 7 is. 1, 2, 7, so 1, 2, 3 took this branch 1, 2, 7 took this branch, fine. So, we seem to be doing fine and then what else is there? Then it is the same thing as this path. The fourth path is very similar to the second path. What is the difference? The only difference is that 1, 2, 3, 5 here 1, 2, 3, 4. So, let us see what does that do; 1, 2, 3, 4, 2, 3, so it does cover this part of the second decision statement and similarly this covers the other half of the second decision statement through 4 and this covers the other half of the second decision statement through 5.

So, the question that was asked was; does it, which coverage criteria does it use? All-defs, All-uses or All du-paths. The correct answer is that it does all uses coverage. You could have also said All-defs coverage, but please remember All-uses coverage subsumes all defs coverage. So, we are looking for the stronger option here, the stronger correct answer is All-uses coverage. So, if you had written All-defs coverage, then marks would not have been given unless you had written All-uses coverage as your answer.

It does not cover All du-paths. Can you tell me which is the du path that it is going to leave out? It will be good to look at this control flow graph and help yourself by understanding which is the du-path that is left out and why do these set of test paths not satisfy all du-paths coverage. Please do that as a small little self check for yourself. So the correct answer to this question would be All-uses coverage.

(Refer Slide Time: 17:38)



Question 6(d)

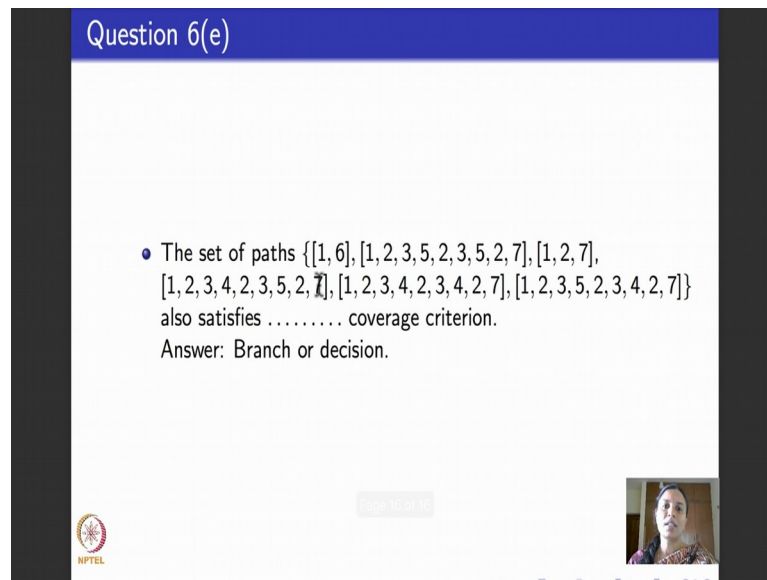
- There is a definition of c in node 5. This definition will be used if the path is included.
Answer: [5,2,3].

NPTEL

The next question asks; tells you there is a definition of c in node 5. So we will go back to the control flow graph and see if there is a definition of c in node 5? Yes, because c is defined given this assignment statement, so c is defined. Then, it asks you to find out where c is used? How will c be used? The only way to use c would be to go out of 5 and the only way to go out of 5 would be to go back to 2. So, once I go back to 2, I have a choice: I could go to 7 or I could go to 3. But my goal is to be able to see this definition of c where was it used; that was the specific question about.

So, from 5 I go back to 2 and suppose I take the path 2, 7, clearly this path does not result in the use of c , because the use of this edge at d and e and the use of this vertex 7 is x . So, this does not help for that and suppose I take 3; if you see c is used here to define t . A correct answer to that question would be; the definition of c will result in a use if the path [5, 2, 3] is included. You have to be able to do 5, 2 and then go to 3; is that clear?

(Refer Slide Time: 18:51)



The screenshot shows a presentation slide with a blue header bar containing the text "Question 6(e)". The main content area is white and contains a bullet point: "• The set of paths {[1, 6], [1, 2, 3, 5, 2, 3, 5, 2, 7], [1, 2, 7], [1, 2, 3, 4, 2, 3, 5, 2, 7], [1, 2, 3, 4, 2, 3, 4, 2, 7], [1, 2, 3, 5, 2, 3, 4, 2, 7]} also satisfies coverage criterion. Answer: Branch or decision." In the bottom right corner, there is a small video feed of a person. The NPTEL logo is visible in the bottom left corner of the slide.

So, the next question again gives you a whole set of paths and it asks you which is the coverage criteria that it satisfies. A simple answer to this question would be branch coverage. You could have also written decision coverage, that would have also been correct. Why does it cover branch coverage? Because if you see this set of paths nicely take the three branches corresponding to three if statements and these set of paths precisely take those branches. This path takes the first branch [1, 6]; then these are the paths out of the second branch 1, 6 and 1, 2 and in 1, 2; you could do [1, 2, 3] or [1, 2, 7] which are the two options and 3; you could do [1, 2, 3, 4] or [1, 2, 3, 5].

So, we will go back to the CFG for a minute, so I do 1, 6 or 1, 2 then at 2; I have a choice [1, 2, 7] or [1, 2, 3] for the two branches and then at 3; I have a choice [1, 2, 3, 4] or [1, 2, 3, 5]. So, all possible branches are listed here and the set of paths that this coverage criteria satisfy; would be branch coverage or decision coverage. You should have got full marks, if you had written either of these options.

So, I hope this assignment helped you to understand how look at simple programs, draw their control flow graphs, annotate it with defs and uses and compute data flow and structural coverage criteria over them. What we will do from next lecture onwards would be to look at logical predicates, coverage criteria over logical predicates and read all these exercises all over again. Take source code, see how logic applies to source code, take specifications, and see how logic applies to specification and so on.

Thank you.

Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

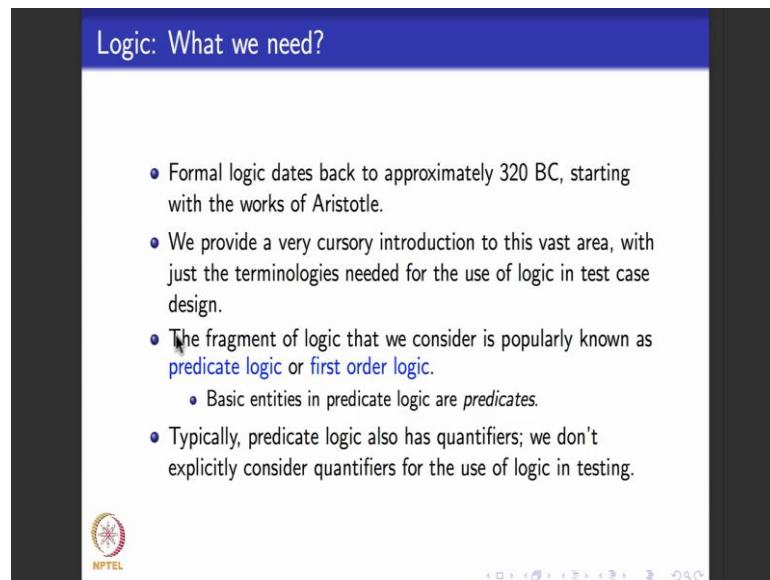
Lecture - 21
Logics: Basics needed for Software Testing

Hello again. We are in week 5: finally, done with graph coverage criteria. What we are going to do now is to move on and see algorithms for test case design based on predicates and logic. You might ask why logic? Why are we looking at logic? So the reason is, if you see a typical program, every kind of decision statement in the program-- - if, while, for loops, do while loops and so on; they have a predicate in them and the predicate is an expression that is meant to evaluate to true or false, and based on whether it evaluates to true or false, different execution paths are taken by the program.

So, logic is the very important part of the program and this week, what we are going to see is how to design test cases based on the logical predicates that occur in programs and later based on the logical predicates that occur as a part of specifications. So, like we did for graph coverage criteria, we will not look at programs and predicates that occur in them first; instead what we will spend time on, is directly looking at logic.

In this module, I will give you a basic of logic, basic introduction to logic as we would need it for design of test cases. Next module, we will introduce coverage criteria based on logical predicates without really seeing where these logical predicates come from. And then after seeing the coverage criteria, the different kinds, their subsumption from a purely theoretical basis, we will go and look at how we can apply to do the coverage criteria on source code and then on specification.

(Refer Slide Time: 01:57)



Logic: What we need?

- Formal logic dates back to approximately 320 BC, starting with the works of Aristotle.
- We provide a very cursory introduction to this vast area, with just the terminologies needed for the use of logic in test case design.
- The fragment of logic that we consider is popularly known as **predicate logic** or **first order logic**.
 - Basic entities in predicate logic are *predicates*.
- Typically, predicate logic also has quantifiers; we don't explicitly consider quantifiers for the use of logic in testing.

NPTEL

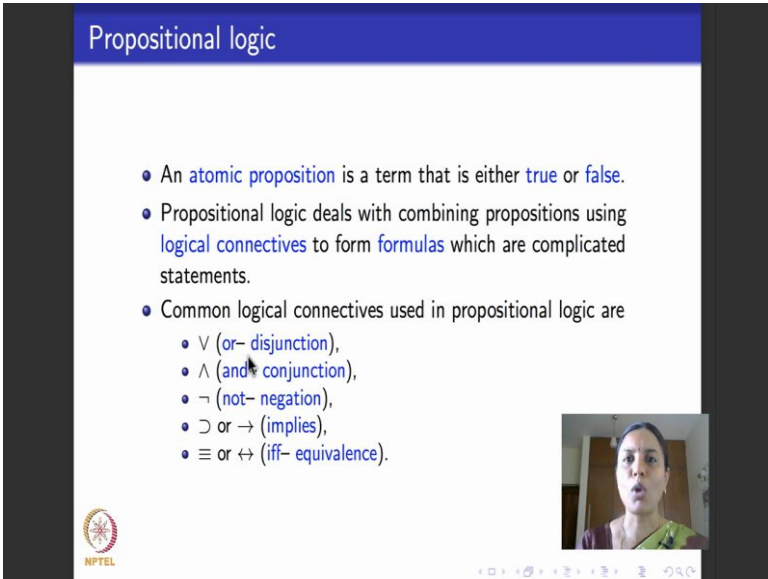
So in this module, I will introduce you to the basics of logic as we would need it for software testing. Formal logic is a very old area, as old as Mathematics is. Philosophers, mathematicians and astronomers several people have used logic. It approximately dates back to 320 BC starting with the work of Aristotle. By no means I can do justification to introducing you to any kind of decent fragment of logic in this course. The goal of this lecture is to be able to look at logic, just about as it is enough for designing test cases as we would need in this course in software testing.

So, the fragment of logic that we need to work with when we do software testing is what is called predicate logic or first order logic. Here you assume that there are functions, relations, put together called as predicates and then they are meant to eventually assume true or false values. But to be able to get to predicate logic, I need to introduce what is called propositional logic which most of you; if you have done a course in discrete maths or a course in logic, you would know about propositional logic. We need propositional logic which is a basic building block of every other logic including predicate logic and then we will do predicate logic.

Typically, another thing to note is that predicate logic or first order logic also has quantifiers. You might have heard two quantifiers for all and there exists, its very important part of first order logic or predicate logic, but as far as its use in testing is concerned we do not really need these quantifiers. So, I will introduce predicate logic

assuming that they are not going to use these quantifiers explicitly. Before we get on to predicate logic, I would like to spend some time recapping propositional logic. As I am not really sure if each of you have been through a course on discrete maths and really know propositional logic. This assumes that you have not seen it and we will do it from the basics. I will introduce you to the basics of propositional logic as we need it and then move on to predicate logic.

(Refer Slide Time: 03:58)



Propositional logic

- An **atomic proposition** is a term that is either **true** or **false**.
- Propositional logic deals with combining propositions using **logical connectives** to form **formulas** which are complicated statements.
- Common logical connectives used in propositional logic are
 - \vee (**or**- disjunction),
 - \wedge (**and**- conjunction),
 - \neg (**not**- negation),
 - \supset or \rightarrow (**implies**),
 - \equiv or \leftrightarrow (**iff**- equivalence).

NPTEL

So, what is propositional logic? Propositional logic can be thought of as a absolute basic logic that occurs as a subset of several different kinds of logic that are used in mathematics and philosophy. The building blocks of propositional logic are what are called atomic propositions. An atomic proposition can be thought of as a Boolean entity or a Boolean variable. It is always meant to be either true or false. and propositional logic tells you how to take an entity that is true or false, that is how to take an atomic proposition and how to combine it with the Boolean connectives.

So, here are the various Boolean connectives that we would be using or written like this, written like a \vee and a conjunction written like this; read it as; and not or negation written like this and implies could be represented in two different ways, this is right arrow symbol and this could be thought of as a superset symbol. Different books use these symbols interchangeably. So, that is an implication operator and then finally, you have

an equivalence operator which is again written in two different ways, you have these three lines or you have a double headed arrow.

So propositional logic, building blocks are propositions then uses these logical connectors to combine and talk about combinations of propositions.

(Refer Slide Time: 05:22)

Atomic propositions

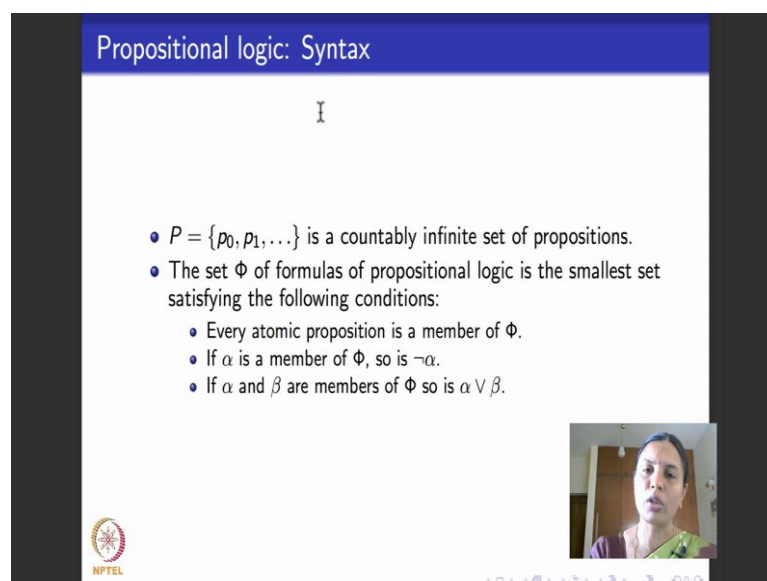
- Propositions are the basic building blocks of logic.
- An atomic proposition or just a proposition is a declarative sentence that is either true or false but not both.
- Examples of propositions:
 - New Delhi is the capital of India.
 - $2 + 3 = 5$.
 - $3 + 3 = 8$.
 - Today is Friday.

NIPTEL

So, what are propositions? As I told you they are basic building blocks of propositional logic. An atomic proposition or a proposition is just an entity that is either true or false and there is clarity about when it is true and when it is false. But at no point in time, it can both be true or false and at no point in time, it can neither be true nor be false. So, here are some simple examples of atomic propositions, so the sentence which says; New Delhi is the capital of India, is an atomic proposition and we know that because Delhi is the capital of India; this proposition is true. And similarly, this statement which says $2 + 3 = 5$; that is if you add 2 and 3 you get 5 is an atomic proposition because it always evaluates to be true or false. In this case it evaluates to be true.

The next statement $3 + 3 = 8$; another statement about addition is another atomic proposition, but in this case the atomic proposition is false because we know that in the world of numbers that we deal with $3 + 3$ is not equal to 8. And the last one, today is Friday is another atomic proposition. It is true every Friday and on days that are not Fridays, it is false.

(Refer Slide Time: 06:33)



The slide is titled "Propositional logic: Syntax" in a blue header. Below the title, the letter 'I' is centered. A bulleted list follows, defining the syntax of propositional logic. The first bullet states that $P = \{p_0, p_1, \dots\}$ is a countably infinite set of propositions. The second bullet states that the set Φ of formulas of propositional logic is the smallest set satisfying three conditions: every atomic proposition is a member of Φ ; if α is a member of Φ , then $\neg\alpha$ is also a member; and if α and β are members of Φ , then $\alpha \vee \beta$ is also a member. In the bottom right corner, there is a small video inset showing a woman speaking. The NPTEL logo is in the bottom left corner.

- $P = \{p_0, p_1, \dots\}$ is a countably infinite set of propositions.
- The set Φ of formulas of propositional logic is the smallest set satisfying the following conditions:
 - Every atomic proposition is a member of Φ .
 - If α is a member of Φ , so is $\neg\alpha$.
 - If α and β are members of Φ so is $\alpha \vee \beta$.

So, in propositional logic, we begin with what is called as syntax of propositional logic. So, what is syntax of propositional logic? It assumes that we have a set of propositions written like this P as p_0, p_1, p_2 and so on. What sort of a set is this? We assume that the number of propositions that are available to this set for us is infinite and countably infinite. Countably infinite means it can take the propositions and enumerate them one after the other.

So, I have as many propositions as I need for use. So we begin with set of countably infinite propositions, typically represented by p_0, p_1, q, r and so on and then I define the set of formulas of propositional logic. What are the set of formulas for propositional logic? It is a smallest set that contains every atomic proposition and inductively, if it contains a formula α ; then it also contains negation of α , read this as $\neg \alpha$. If it contains formulas α and β , then it also contains $\alpha \vee \beta$. So, this is how I define all the formulas of propositional logic. It so turns out that these three entities; the atomic propositions, negation operation and disjunction operation or “or” are enough to define all the formulas of propositional logic.

So, you might wonder that I gave all these as also connectors and implies if and only if, but when I defined the syntax here, we use only not and or. What happened to the rest? The rest are what can be derived from using not and or operators.

(Refer Slide Time: 08:17)

Derived operators

- And: $\alpha \wedge \beta: \neg(\neg\alpha \vee \neg\beta)$
- Implies: $\alpha \supset \beta: \neg\alpha \vee \beta$
- Lff: $\alpha \equiv \beta: (\alpha \supset \beta) \wedge (\beta \supset \alpha)$.

The slide features the NPTEL logo in the bottom left corner and a small video inset of a presenter in the bottom right corner.

So, and can be written as $\alpha \wedge \beta$ is nothing but negation of not negation alpha or negation beta. $\alpha \wedge \beta$ is nothing but $\neg(\neg\alpha \vee \neg\beta)$; alpha implies beta can be defined as not alpha or beta. Alpha if and only if beta or alpha equivalent to beta can be defined as alpha implies beta and beta implies alpha.

(Refer Slide Time: 08:48)

Propositional logic formulas: Examples

- You cannot ride the roller coaster if you are under 4 feet tall unless you are older than 16 years.
 $(r \wedge \neg s) \supset \neg q$, where q , r and s represent "You can ride the roller coaster", "You are under 4 feet tall" and "You are older than 16 years" respectively.
- Maria will find a good job when she learns Software Testing.
 $p \supset q$, where p and q represent "Maria learns Software Testing" and "Maria will find a good job" respectively.

The slide features the NPTEL logo in the bottom left corner and a small video inset of a presenter in the bottom right corner.

So, moving on, here are some examples of propositional logic formulas. So here is a simple sentence which says that you cannot ride a roller coaster if you are under 4 feet tall or unless you are older than 16 years. So, how do I represent it as a propositional

logic formula? First I have to figure out what the atomic propositions in these entities are. So if you see there are three statements. The first statement can be thought of as this; you cannot ride the roller coaster, second statement is you are under 4 feet tall, third statement is you are older than 16 years.

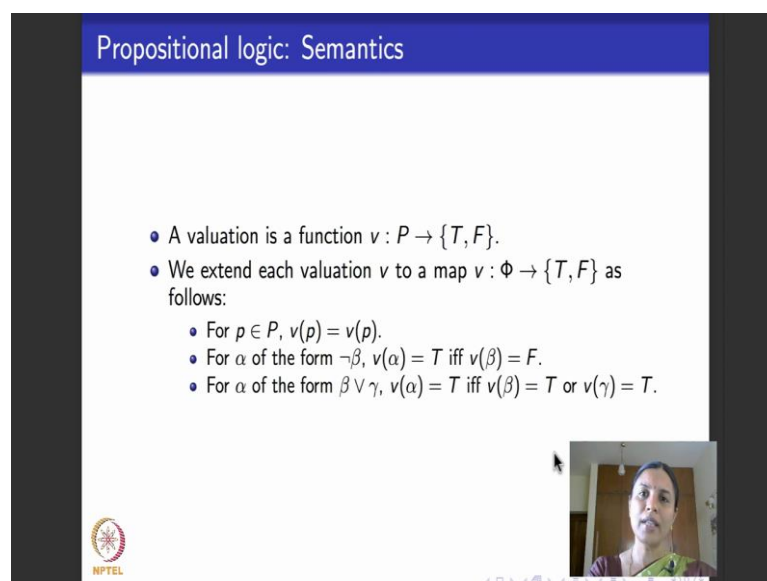
So, give them names, let us say q represents you can ride the roller coaster, you could say $\neg q$ represents you cannot ride the roller coaster. We just change the negation appropriately, there will be no difference. Let us say r represents you are under 4 feet tall; the second phrase and s represents you are older than 16 years; the third phrase. So, now what does this formula say? Take the first phrase which is $\neg q$ which is here; it says you cannot ride the roller coaster if; if is there and you cannot. So, there is an if and a not; if and a not can be this, α implies β , read it as if α then β .

So, if α then β is same as α implies β , so to be able to express you cannot ride the roller coaster if you are under 4 feet tall, I need to use an implication sentence. But then there is also one more atomic proposition here which is being older than 16 years and then there is an unless connective here. Unless means that you have to be older than 16 years, if you are under 4 feet tall to be able to ride a roller coaster. So, if I try to use the connectors that we learnt so far and try to write it as a propositional logic formula, this is the formula that I will get. What does it say? It says that you are under 4 feet tall or and you are older than 16 years and when you are not older than 16 years sorry implies you cannot ride the roller coaster.

So, I will repeat it again you are under 4 feet tall, so, r is true and you are not older than 16 years, which means you are younger than 16 years. These two implies that $\neg q$ is true. What does $\neg q$ say? $\neg q$ says, you cannot ride the roller coaster because we have instantiated q to be you can ride the roller coaster.

So, here is one more example. This example says Maria will find a good job when she learns software testing. So, how many atomic propositions can be created out of this sentence: one atomic proposition is which says Maria will find a good job, the second is if she learns software testing, some Maria learns software testing. So, I say p represents Maria learns software testing, q represents Maria will find a good job, then the sentence p implies q will mean Maria learns software testing if Maria finds a good job.

(Refer Slide Time: 11:44)



Propositional logic: Semantics

- A valuation is a function $v : P \rightarrow \{T, F\}$.
- We extend each valuation v to a map $v : \Phi \rightarrow \{T, F\}$ as follows:
 - For $p \in P$, $v(p) = v(p)$.
 - For α of the form $\neg\beta$, $v(\alpha) = T$ iff $v(\beta) = F$.
 - For α of the form $\beta \vee \gamma$, $v(\alpha) = T$ iff $v(\beta) = T$ or $v(\gamma) = T$.

NPTEL

So, now syntax tells you how to write formulas, how to build formulas. Now, what is the semantics of propositional logic. What does semantics mean? Semantics mean what is the meaning of these formulas. For example, if I go back and look at these two formulas r and not s implies not q and I look at p implies q , how do I know what they mean? Are the formulas true, are the formulas false, how will I know? How will I infer that? That is what semantics is all about.

So, semantics, we begin with a valuation function which takes every atomic proposition P and tells me whether it is true or false. So, valuation is the function from the set P of atomic propositions to the sets T and F , where T stands for true and F stands for false. Throughout this lecture, in the lecture for logic, I will use the term T for true and F for false. These are Boolean constants. Now, we extend valuation what other parts were there in the syntax of propositional logic? Every atomic proposition was there and then all these operators--- negation, disjunction and then derived operators conjunction, implication and equivalence.

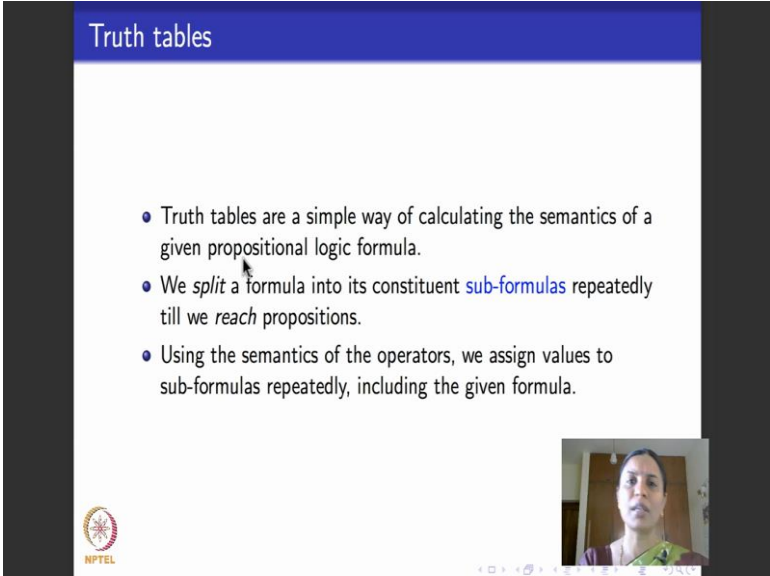
So, once I define, once I start with the valuation function that tells me whether every atomic proposition is true or false, I can go ahead and extend the valuation function to the set of all formulas Φ and it make each formula of Φ become true or false. So, extension of valuation is a map v that takes a set of all formulas Φ and each formula ϕ

whether it tells true or false by extending the valuation function v that takes the proposition and tells me whether each proposition is true or false. How is that extended?

So, if α is of the form not β , then v of α is true if and only if v of β is false. So, it says that if α is of the form negation β , then α is true if and only if β is false. If β is false then not β will be true so that is why α is true. Now if α is of the form β or γ then, α is true if and only if β is true or γ is true. Please note that this is not an either or statement, it is just a plain or. That means, that if both β and γ turn out to be true, then α can be true also. The only time when α is false is when both β and γ become false.

So, I have not given you the semantics of other three operators. But it is easy to infer what they are because we have given semantics of not and or and other three operators can be defined using them. Or, you could directly write the semantics of these operators. For example, when will α and β be true in under valuation function? If both α and β , both individually evaluate to true.

(Refer Slide Time: 14:33)



The slide is titled "Truth tables" in a blue header. It contains three bullet points:

- Truth tables are a simple way of calculating the semantics of a given propositional logic formula.
- We *split* a formula into its constituent *sub-formulas* repeatedly till we *reach* propositions.
- Using the semantics of the operators, we assign values to sub-formulas repeatedly, including the given formula.

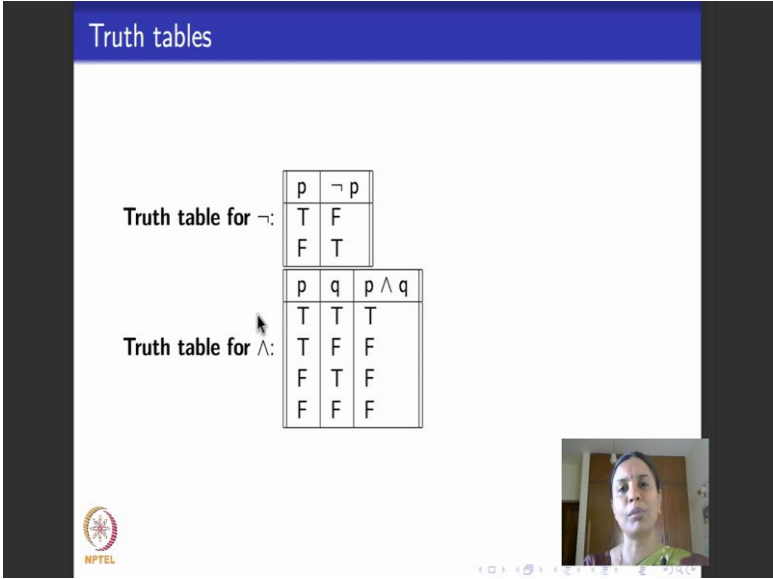
In the bottom right corner, there is a small video inset showing a woman speaking. The NPTEL logo is visible in the bottom left corner of the slide.

So, later very soon I will tell you what truth tables are and then we will see what each of these semantics mean in terms of truth tables. Some of you might find this notation a bit cumbersome: to read valuation as a function from the set of propositions to true and false. But most of you might be familiar with the notion of truth tables. What are truth

tables? Truth tables are a simple way of calculating the semantics of a propositional logic formula.

So, what we do is we take a formula and we split the formula or break the formula into its constituent sub formulas repeatedly till we reach what are called atomic propositions, and when I reach atomic propositions, I have my valuation function which tells me when each proposition will become true or false. Then, I use the semantics of the Boolean operators and of the negation operators and work my way up till I know whether the entire formula is true or false. I have put these words split and reach in italics because if you have to do it properly, then you need to be able to parse the formula, generate the parse tree and the truth table works inductively bottom up beginning from the leaf all the way till the root of the parse tree which contains the formulas.

(Refer Slide Time: 15:46)



Truth tables

Truth table for \neg :

p	$\neg p$
T	F
F	T

Truth table for \wedge :

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

So, what are the various truth tables for the elementary connectors of Boolean logic? So, here are the truth tables. How does the truth table for not look like? Remember the valuation for the negation function not, if p is true, not p is false and if p is false, not p becomes true; that is what this table says. How does the truth table for and look like? For and it is a binary connective, so it takes two operands p and q. When is p and q true? p and q is true, if both p and q are true. That is what this first row says, if p is true and q is true then p and q is true. In all other places, p could be true, q could be false, p could be

false, q could be true, both p and q could be false, in all the other three cases p and q evaluates to be false because one of p or q is false.

(Refer Slide Time: 16:37)

Truth tables		
Truth table for \vee :	p	q
	$p \vee q$	
	T	T
	T	F
Truth table for \supset :	p	q
	$p \supset q$	
	T	T
	T	F
Truth table for \equiv :	p	q
	$p \equiv q$	
	T	T
	T	F

So, for or, we saw the semantics of valuation function p or q is true; if one of them is true or if both of them are true. So, if p is true, q is true p or q is true that is this first row; if p is true; q is false p or q is true that is second row; if p is false and q is true, p or q is again true because in all three cases one of p or q is true. But, when both p and q are false, p or q becomes false again. What is the truth table for implication? If you go back to the slide which defines the meaning of implication, it says α implies β is defined as not α or β . Or if you want to understand it in English; read α implies β as if α then β .

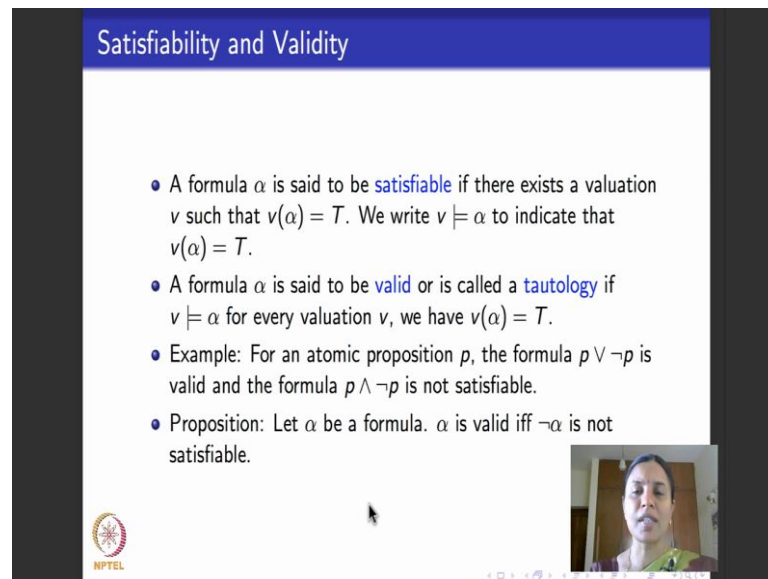
So, if I use that meaning then this is the truth table for implication. If p is true then and q is true then, obviously p implies q is true, because p is true q is also true, so this is true. So, if p is true and q is false; then p implies q will be false because it cannot be the case that p is true and p implies q will be true when q is false. If p is false then we do not really worry whether q is true or false, we say p implies q is true in a trivial sense. So, when both the cases when p is false and q is true or false, we say p implies q is true trivially.

What is the truth table for equivalence? The truth table for equivalence says that p equivalent to q is true if p and q have the same truth value. That is they both are true

together, which is this first row of the table or they both are false together, which is the last row of the table.

The second row and the third row, one of p or q turn out to be false, so p is not equivalent to q . In other words p equivalent to q itself turns out to be false.

(Refer Slide Time: 18:43)



Satisfiability and Validity

- A formula α is said to be **satisfiable** if there exists a valuation v such that $v(\alpha) = T$. We write $v \models \alpha$ to indicate that $v(\alpha) = T$.
- A formula α is said to be **valid** or is called a **tautology** if $v \models \alpha$ for every valuation v , we have $v(\alpha) = T$.
- Example: For an atomic proposition p , the formula $p \vee \neg p$ is valid and the formula $p \wedge \neg p$ is not satisfiable.
- Proposition: Let α be a formula. α is valid iff $\neg\alpha$ is not satisfiable.

The slide also features a small video inset of a person in the bottom right corner and a logo in the bottom left corner.

Now, we move on to the notions of satisfiability and validity. Why are these important? These are important because I told you that propositional logic formulas or predicate logic formulas are going to come as labels of decision statements in programs and when I have a predicate as a label of a decision statement in a program, I am evaluating the predicate by substituting some values for the variables that occur in the predicate and I am checking whether the predicate becomes true or false. If it becomes true, then the decision statement takes one path, if it becomes false then the corresponding decision statement takes another path.

So, in logic we call this as the problem of satisfiability. The problem of satisfiability involves checking whether a given logical formula evaluates to true or not. So, for propositional logic also, the problem of satisfiability checks whether given a formula α , is there a valuation function v such that v makes α true? That is v of α is true; in logic we write like this, we write $v \models \alpha$ read this entity as v satisfies α to indicate that v of α is true. Please read this notation as v satisfies α .

The notion contrapositive to satisfiability is what is called validity you might have heard about it. We say that a formula is valid, if for every valuation v it becomes true. So we say α is valid if no matter what truth values you assign to the atomic propositions in α , α always becomes true. Another term for valid formula is what is called a tautology. The exact opposite of valid formulae are what are called contradictions, which mean that no matter what truth values you assign to atomic formulas, the formula will never be satisfiable; that is it will always be false.

So, here is a simple example of a formula that is valid and of a formula that is a contradiction. Consider an atomic proposition p . A formula of the form p or not p is always valid, why? Because if p is true then not p will be false, but this is a or, so the whole thing will be true. On the other hand, if p is false then not p will become true, again because this is a or, the whole thing becomes true. So, the p or not p is a formula that will always become true irrespective of whether p is true or false.

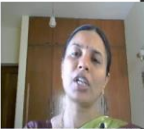

Now, consider a formula of the form p and not p . If you see here p and the not p will never be satisfiable, no matter what p is because if p is true then not p will become false and because it is an and here; the whole thing will be false. Similarly, if p itself is false then because it is an and here p and not p will be false. So, no matter what p is whether it is true or whether it is false; p and never p ; p and not p will always be a contradiction.

Here is a small result. It says that consider a formula α ; α is valid if and only if not α is not satisfiable. So, it says validity and satisfiability are what are called contrapositives of each other. The proof is very simple, but we would not really need it for this course, so I am leaving the proof without doing it. So, the only concept that you need to remember mainly from this slide for the rest of the course is to understand what satisfiability is. We say a formula is satisfiable if there is at least one valuation for the atomic propositions in the formula that make the entire formula true. Typically, the most common way of checking satisfiability is to be able to use truth tables.

(Refer Slide Time: 22:24)

Satisfiability problem of propositional logic

- To check if a formula α is satisfiable, construct the truth table for α and check if there is an entry (valuations of propositions) that makes α true.
- This algorithm takes time exponential in the length (number of symbols) of α .
- Satisfiability problem for propositional logic is NP-complete.





(Refer Slide Time: 22:29)

Satisfiability through truth tables: Example

Truth table for the formula $(r \wedge \neg s) \supset \neg q$.

r	s	q	$r \vee s$	$(r \vee s) \vee \neg q$
T	T	T	T	T
T	T	F	T	T
T	F	T	T	T
T	F	F	T	T
F	T	T	T	T
F	T	F	T	T
F	F	T	F	F
F	F	F	F	T



So, here is an example of how to check satisfiability using truth tables. So, you consider this formula r or s or not q , so here is a truth table for this formula. So, what I have done is, I have given true, false assignment to all these values. So, there are three atomic propositions here. So if I consider the possible combinations of true, false values to each of them, there will be 8 different combinations 2^3 is 8. So, r , s and q all three of them take true or r and s become true, q becomes false; r is true, s is false, q is true and so on. Now, I will first evaluate r or s which is, I use the semantics only for these two

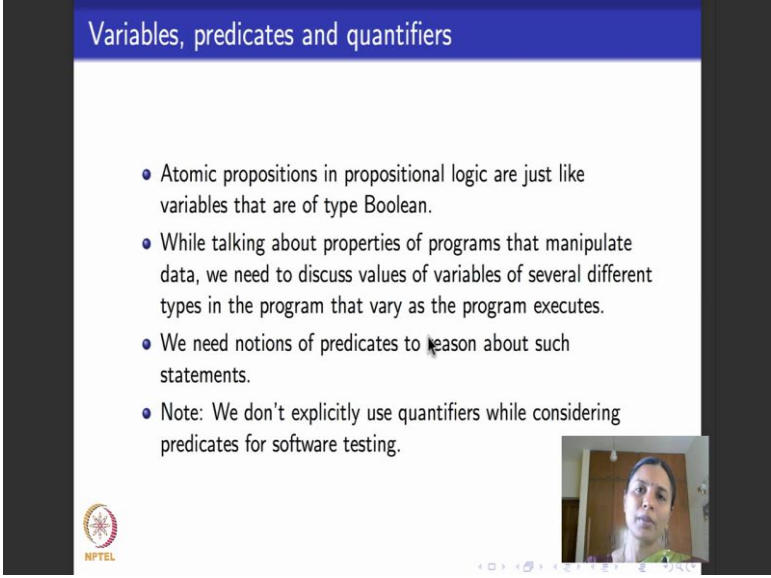
rows and the semantics of r to fill up this row with true false values. If you see in the first six rows, one of r or s will always be true here, so r or s throughout becomes true.

The last two rows both r and s are false, so r or s is false. Now, I do this r or s or not q ; I take r or s , then I take the row the column for q ; negate the column which I have not shown in the truth table here. So, if you want to be perfect, you could negate this column and then I apply or again. If I do that; then I will get all these values to be true and this value to be false, so this is how truth tables work.

Now, to check if a formula α is satisfiable, what I do is I generate the truth table and I check if there is at least one row in the last column of the truth table which corresponds to the formula, where the formula evaluates to be true. If there is one such row, then it means there is an assignment of true-false values to the variables of the formula that make the formula true. So in which case the formula is satisfiable. So if you see what is the running time of such an algorithm? I have not really described the algorithm.

But what is the running time of such an algorithm? The running time of this algorithm is going to be exponential in the number of symbols of α because I told you; suppose there are three different variables, then each of them will take two different truth values and the number of combinations is going to be 2^n . So, the number of rows in the truth table in the worst case is going to be 2^n and so the algorithm runs in exponential in the number of symbols or the number of atomic propositions within. In general, it is known that the satisfiability problem for propositional logic is NP-complete. So, NP-complete means we do not know of a polynomial time algorithm till date that we can use to solve satisfiability.

(Refer Slide Time: 25:12)



Variables, predicates and quantifiers

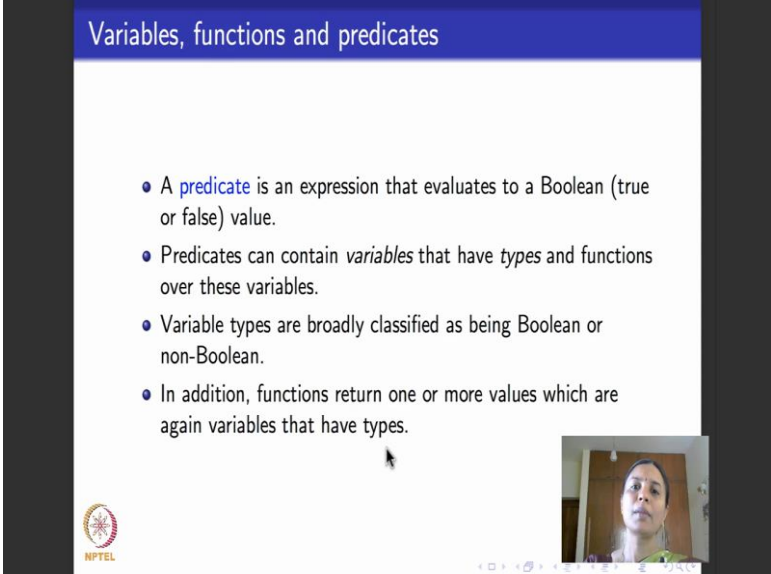
- Atomic propositions in propositional logic are just like variables that are of type Boolean.
- While talking about properties of programs that manipulate data, we need to discuss values of variables of several different types in the program that vary as the program executes.
- We need notions of predicates to reason about such statements.
- Note: We don't explicitly use quantifiers while considering predicates for software testing.

NPTEL

Now, that was a basic introduction to propositional logic. But as I told you in the beginning of this module, what we would need is what is called predicate logic. What are predicate logics? Predicate logics are used to define predicates which come as labels of decision statements and programs. Predicates have variables of all different kinds. There could be integer variables, they could be floating point variables, there could be functions evaluating certain things. All of them need not be just Boolean variables like in propositional logic.

So, we need to be able to move on. So, atomic propositions and propositional logic just define Boolean entities. But, when we talk about programs that manipulate data, we encounter other kinds of variables, so we need notions of predicates to reason about such statements. As I told you in the beginning of this lecture, strictly speaking, predicate logic also deals with quantifiers for all and there exists, but we will not need that for testing; so I am staying away from introducing them to you.

(Refer Slide Time: 26:13)



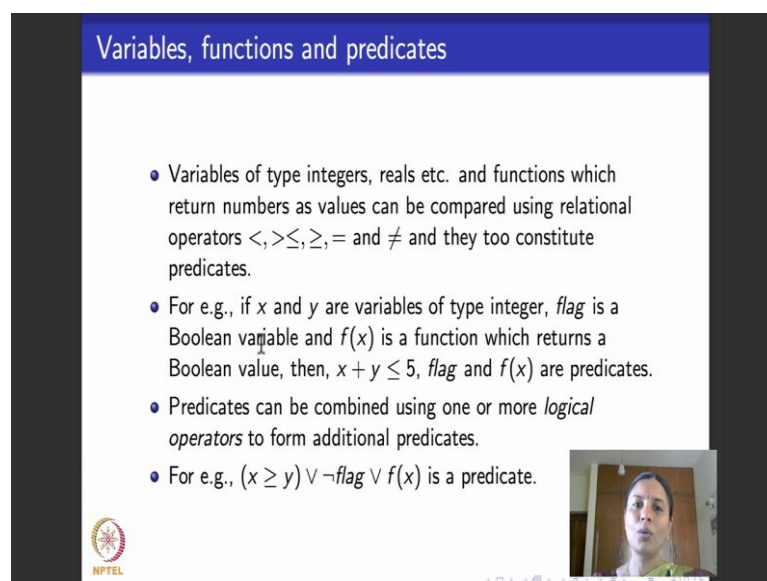
The slide has a blue header with the text "Variables, functions and predicates". Below the header, there is a list of four bullet points. In the bottom right corner, there is a small video inset showing a woman speaking. In the bottom left corner, there is a small logo for NPTEL.

- A **predicate** is an expression that evaluates to a Boolean (true or false) value.
- Predicates can contain *variables* that have *types* and functions over these variables.
- Variable types are broadly classified as being Boolean or non-Boolean.
- In addition, functions return one or more values which are again variables that have types.

So, what is a predicate? For our purposes a predicate can be thought of as an expression that always evaluates to true or false; because that is what we need as far as the course is concerned. Now, predicates can contain variables like we find them in programs and each variable could have different type. There could be integer type variables, there could be strings, there could be floating point numbers. Predicates also can contain functions. Like for example, you would agree with me that it is not very uncommon to see a statement which says that if \log of x is less than 0.1 then you do something.

So, what is \log of x ? \log of x is a function that takes x and evaluates \log of x and returns a number. So, predicates can contain function that return values of a certain type. We broadly classify variable types as Boolean and non Boolean because all non Boolean entities will be of one type as far as our semantics of predicates are concerned and functions return one or more values, which are again variables that have types.

(Refer Slide Time: 27:16)



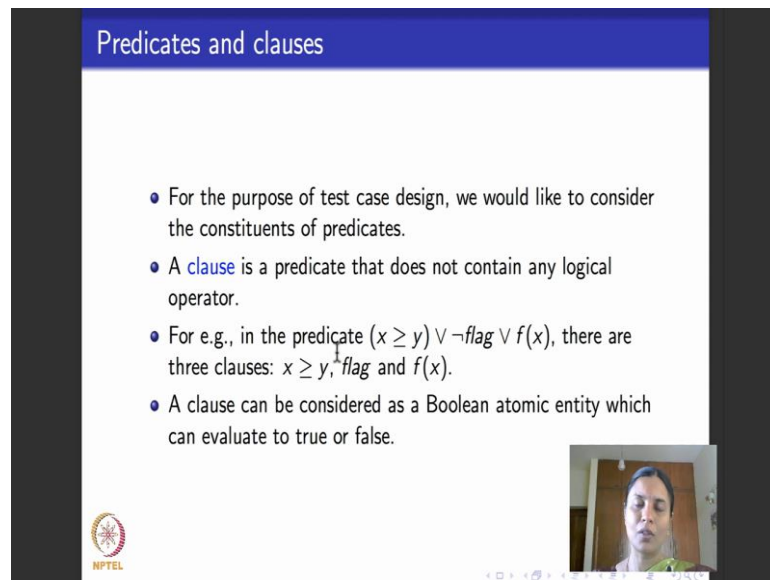
The slide is titled "Variables, functions and predicates" in a blue header. It contains a list of four bullet points. The first bullet point states that variables of type integers, reals, etc., and functions which return numbers as values can be compared using relational operators $<$, $>$, \leq , \geq , $=$, and \neq , and they too constitute predicates. The second bullet point gives an example: if x and y are variables of type integer, $flag$ is a Boolean variable and $f(x)$ is a function which returns a Boolean value, then $x + y \leq 5$, $flag$ and $f(x)$ are predicates. The third bullet point states that predicates can be combined using one or more logical operators to form additional predicates. The fourth bullet point gives an example: For e.g., $(x \geq y) \vee \neg flag \vee f(x)$ is a predicate. In the bottom right corner, there is a small video inset showing a woman speaking. The NPTEL logo is in the bottom left corner.

- Variables of type integers, reals etc. and functions which return numbers as values can be compared using relational operators $<$, $>$, \leq , \geq , $=$ and \neq and they too constitute predicates.
- For e.g., if x and y are variables of type integer, $flag$ is a Boolean variable and $f(x)$ is a function which returns a Boolean value, then, $x + y \leq 5$, $flag$ and $f(x)$ are predicates.
- Predicates can be combined using one or more *logical operators* to form additional predicates.
- For e.g., $(x \geq y) \vee \neg flag \vee f(x)$ is a predicate.

Now, you might have seen that in predicates that we use in programs, you would have used all these operators, so called relational operators. So, variables of types integers, real numbers and so on and functions which return numbers as values can be compared using the normal relational operators and numbers; less than, greater than, lesser than or equal to, greater than or equal to, equal to, not equal to and so on.

Like for example, if I have x and y as variables of type integer, $flag$ is a Boolean variable and f of x is a function which returns the Boolean value, then here are some examples of predicates. I can ask whether x plus y is less than or equal to 5; this entity x plus y will give me a number; 5 is another number; the whole predicate x plus y less than equal to 5 will return true or false. $flag$ itself is a Boolean variable; so it is true or false; f of x is a function which returns a Boolean value which is again true or false. So, all these are predicates. And, like we saw in propositional logic, each of these predicates can be combined using one or more logical operators. Like for example, if I had these predicates then here is a predicate that combines together. So, x is greater than equal to y or not a $flag$ or f of x . And the whole thing will evaluate to true or false because each of these entities will evaluate to true or false and I can use the semantics of the or operator to evaluate the meaning of the entire predicate.

(Refer Slide Time: 28:53)



The slide is titled "Predicates and clauses" in a blue header. It contains four bullet points:

- For the purpose of test case design, we would like to consider the constituents of predicates.
- A **clause** is a predicate that does not contain any logical operator.
- For e.g., in the predicate $(x \geq y) \vee \neg flag \vee f(x)$, there are three clauses: $x \geq y$, $flag$ and $f(x)$.
- A clause can be considered as a Boolean atomic entity which can evaluate to true or false.

In the third bullet point, the expression $(x \geq y) \vee \neg flag \vee f(x)$ is shown. The terms $x \geq y$, $flag$, and $f(x)$ are underlined. Arrows point from these underlined terms to the word "clauses" in the text. A small video inset in the bottom right corner shows a person speaking. The NPTEL logo is in the bottom left corner.

For the purposes of this course, to be able to define coverage criteria, I would need one more terminology. What we say is that each individual entity that comes without a Boolean operator in a predicate, we will call it a clause.

So, clause is a predicate that does not contain any logical operator. So, if I have a predicate that looks like this: x greater than equal to y or not $flag$ or f of x , then it has three clauses; one is x greater than equal to y , the other one is $flag$. Alternatively, you could say not $flag$ is also a clause, not a problem or you could say f of x . For our purposes, we say it does not contain any logical operator. So, we remove the not and just say a $flag$ is a clause. So a clause can be thought of as Boolean atomic predicate which always evaluates to true or false.

(Refer Slide Time: 29:45)

Satisfiability problem for predicate logic

- We say that a given predicate p is satisfiable if there is an assignment of values to its constituent clauses (in turn, variables and functions) that make p true.
- Satisfiability problem for predicate logic is undecidable.
- There are *SAT solvers* and *SMT solvers* that can check if a given predicate is satisfiable or not for many different kinds of predicates.

NPTEL

So, in the next lecture what we will do is, we will see how to define coverage criteria based on predicates and clauses. So, before I finish today's lecture, like we saw satisfiability problem for propositional logic, we also consider satisfiability problem for predicate logic for the same reason, because these predicates are going to come as labels of decision statements in the program and decisions in the program are taken based on whether these predicates are true or false.

Typically a program is evaluated given a set of values. A generalization of that problem is to ask given an arbitrary predicate is there at least one assignment of true, false values or is there at least one assignment of values to the variables of the corresponding types that make the predicate true or false. This is what is called as satisfiability problem for predicate logic. Unlike propositional logic, propositional logic we just assign true false values, build our way up using truth tables. You cannot do that for predicate logic because variables could be of different types and if it is a variable like integer then there are potentially an infinite number of values for which you have to check.

So, satisfiability problem for predicate logic is known to be undecidable. There are no algorithms in the general case that solve the satisfiability problem for predicate logic, but there what are called SAT solvers or SMT solvers that help you to do it. Please remember that when we use it in testing, even though we consider satisfiability problem, we are not looking for really one satisfying assignment. As and when program executes,

we check if the given assignment of values to the variables makes the predicate true. So, that is not really the satisfiability problem, but it is useful to know that the satisfiability problem for propositional logic is tough, it is NP-complete and the satisfiability problem for predicate logic in general is undecidable.

So in the next lecture, I will introduce you to coverage criteria based on predicates and clauses.

Thank you.

Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
Indian Institute of Information Technology, Bangalore



Lecture - 22
Logic Coverage Criteria

Hello again, we are in the fifth week, this is the third lecture. Last class I introduced you to the basics of logic, we saw propositional logic, predicate logic. I gave you propositional logic mainly as a recap and as an entity to introduce the logical connectors; negation, conjunction, disjunction, implication and equivalence. The logic that we will be using throughout logic based testing would be predicate logic. So, today what we will begin with is to understand various coverage criteria in terms of logic like we did for graphs now we will focus on logic.

(Refer Slide Time: 00:44)

Re-cap: Predicates and clauses

- A **predicate** is an expression that evaluates to a Boolean value.
- E.g.: $(x > y) \vee C \vee f(z)$ is a predicate where x, y and z are non-Boolean variables, C is a Boolean variable and $f(z)$ is a function that returns a Boolean constant.
- A **clause** is a predicate that does not contain any logical operators.
- E.g., in the above predicate, there are three clauses:
 $x > y, C$ and $f(z)$.

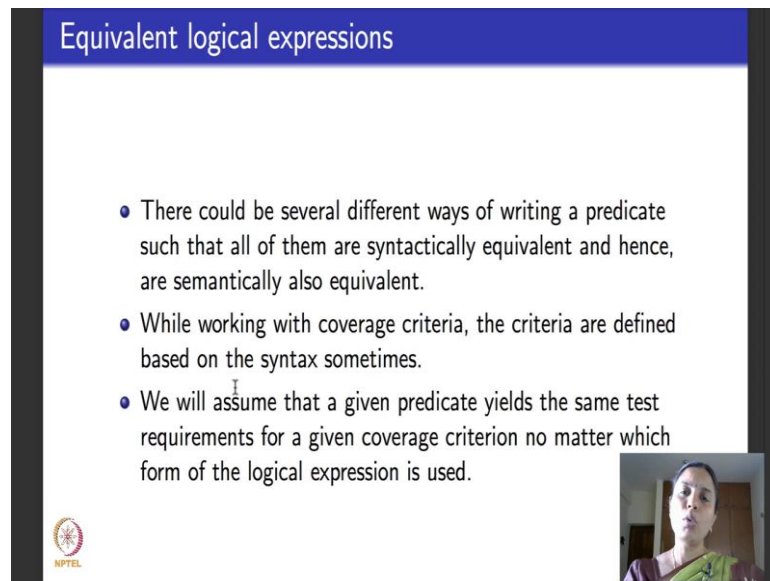


And as I told you, we will work with predicates and clauses. So, I will recap the definition of predicates and clauses that we saw from the last lecture what is a predicate a predicate is any expression any formula that evaluates to a true or a false value.

Here is an example. So, that this is a predicate which has 3 clauses: x is greater than y or C or f of z . x, y and z are some variables, we assume that they are not Boolean. x is greater than y will result in a Boolean value, C is a standalone clause. So, C better be a Boolean entity f of z is directly 'or'ed with these 2 other clauses. So, f of z better be a

function that takes z and return either true or false. So, x is greater than y will be either true or false, C is a Boolean variable, f of z is a function that returns a Boolean value. So, I can 'or' these 3 and get a true or false value for this predicate. Each of these 3 entities in this predicate are what are called clauses. A clause is a predicate that does not contain any logical operators.

(Refer Slide Time: 02:03)



The slide has a blue header with the text "Equivalent logical expressions". Below the header, there are three bullet points:

- There could be several different ways of writing a predicate such that all of them are syntactically equivalent and hence, are semantically also equivalent.
- While working with coverage criteria, the criteria are defined based on the syntax sometimes.
- We will assume that a given predicate yields the same test requirements for a given coverage criterion no matter which form of the logical expression is used.



In the bottom left corner, there is a small circular logo with the text "NPTEL" below it. In the bottom right corner, there is a small video inset showing a woman with dark hair, wearing a green and purple sari, speaking.

So, now before we move on and look at logical coverage criteria, I would like to spend little time making you notice one point. So, as we saw in the last class and even otherwise you might know from logic that the same formula, well formed formula can be written using several different equivalent ways. And when we look at coverage criteria, the clauses and the way the clauses react could be different for each of the equivalent ways, but what we assume throughout this lectures where we deal with coverage criteria is that we assume that the given predicate will yield the same TR for a given criteria no matter which form of the logical expression is used. For example, I could write a implies b directly using implies or I could write a implies b as $\text{not } a \text{ or } b$, but I assume that these are semantically equivalent and as for as coverage criteria is concerned I fix one of them and carry on.

(Refer Slide Time: 02:58)

Predicates and clauses: Notations

- Let P be a set of predicates and C be a set of clauses in the predicates in P .
- For each predicate $p \in P$, let C_p be the clauses in p .
 $C_p = \{c \mid c \in p\}$.
- $C = \bigcup_{p \in P} C_p$.




So, here are some notations that we will be using for the rest of this lecture and next lecture. We assume that we are dealing with a particular piece of software artifact, could be code or it could be requirement given as a finite state machine and that software artifact has a set of predicates. So, we are dealing with a universal set of predicates that come from a given software artifact. Like capital P be that set and C is the set of all clauses that come in the predicates in p . For each predicate p in P the clauses specific to the predicate small p we denote it by the set C suffix p C_p , right. So, what is the set C suffix p , it is a set of all clauses that occur in the predicate p C this is a set of all clauses in the set of predicates p is the union of all the clauses as they occur in each of the predicates in p . We deal in several logical coverage criteria. The first one that we deal with is the most elementary one it is called predicate coverage.

(Refer Slide Time: 03:51)

Predicate coverage

- **Predicate coverage (PC):** For each $p \in P$, TR contains two requirements: p evaluates to true and p evaluates to false.
- E.g., for the predicate $(x > y) \vee C \vee f(z)$, the two tests that satisfy predicate coverage will be $x = 5, y = 3, C = \text{true}, f(z) = \text{false}$ (the predicate evaluates to true) and $x = 1, y = 4, C = \text{false}, f(z) = \text{false}$ (the predicate evaluates to false).
- Overlap with graph coverage criteria: For a set of predicates associated with branches, predicate coverage is the same as edge coverage.



So, if you see the English sense of this term what is predicate coverage mean ? It means cover the predicate, what does it mean to cover the predicate in turn ? Predicate, as we saw is any formula that results or evaluates to a Boolean value. So, when I cover the predicate I am insisting that I am giving one set of values that make the predicate true and one set of values that make the predicate false. So, predicate coverage says that for each predicate p in this set of universal predicates P , my TR or test requirement contains 2 requirements: one requirement insists that the predicate p evaluates to true and the other says p evaluates to false.

For example, if we take this predicate, x greater than y or C or f of z that we saw in the previous slide this whole thing is one predicate p . I want to be able to do predicate coverage for this predicate p which means my TR says that there are 2 test requirements: one that makes this whole predicate true and one that makes this whole predicate false. Now this is my TR. So, what would be the set of test cases that will satisfy this test requirement. Set of test cases that will satisfy this test requirement will be one for each kind of TR. So, I need to write a set of test cases that will make the predicate true and I will write a set of test cases that make the predicate false. So, remember this is a 'or' predicate. So, it is true if any one of these clauses in this predicate become true and it is false if each of the clauses in this predicate become false.

So, to make it true I can assign values for any of these variables such that one of the clauses become true. For example, here is one assignment of value that will make the predicate true. I have assigned x to be 5 y as 3. So, 5 is greater than 3, x is greater than y, this predicate value is to be true. Remember this is enough to make the entire predicate true, but for complete test case we need to go ahead and assign values for all the other variables also. So, here I have assigned C to be true and f of z to be false. So, combining and substituting back these values you realize that 5 is greater than 3. So, x greater than y will be true; true will be 'or'ed with C which is true which in turn will be 'or'ed with f of z which is false. So, it will be true or true or false which will make the entire predicate p to be true.

Now, I need another test case that will make this predicate false because my goal is to be able to do predicate coverage. So, here is such a test case. I assign x as 1, y as 4. So, x is not greater than y. So, this predicate this clause evaluates to be false and I assign C to be false. So, the second clause also becomes false and I make f such that f of z returns false. So, the third clause also evaluates to false. Since all 3 are false the entire predicate becomes false.

Here is a small observation. Now if you go back and look at the graph coverage criteria that we saw predicate coverage has an overlap with graph coverage criteria. If you remember we saw this edge coverage criteria when we did graph coverage which was also called branch coverage. Assuming that every branch in the graph every node corresponding to a branch in a graph is labeled by a predicate, predicate coverage for that predicate will be the same as doing edge coverage for that branch because there will be one set of test cases that will make the predicate true making it take the then part of the branch and there will be another set of test values that will make the predicate false making it take the else part of the branch. So, there will be 2 edges coming out of the node that corresponds to this predicate and it will mean edge coverage for each of these 2 nodes. So, predicate coverage is the same as edge coverage.

Now what is the disadvantage in predicate coverage? If you go back and see this example x greater than y or C or f of z this predicate coverage tests this entire predicate as a whole. So, and if we look at the test values that we assigned to achieve predicate coverage for that predicate in both these set of test cases in the first one and in the second one we had assigned f of z to be false. So, it so, happens that we have tested predicate

coverage, but we did not really exercise the clause f of z being true. It could have been the case that there could have been an error in the software because the clause f of z was true. So, it is not wise to just do predicate coverage and say I have given true and false values to make the entire predicate true once and to make the entire predicate false once and leave it that because we can do it as we did it in this example without exercising each clause. And you never know, may be the clause that wasn't exercised for that particular value had an error in it.

In this case, it could be the case that if f of z was true may be the software had an error. So its not wise just to do predicate coverage. What we would to move on is the next coverage criteria which is called clause coverage. What does clause coverage say? Clause coverage is abbreviated as CC, it says for each clause in the set of all clauses TR contains 2 requirements--- that clause evaluates to true that clause evaluates to false. So, it says exercise each clause and make it true once make it false once. That is what clause coverage is said to do. So, if you see this example, it had 3 clauses: x greater than y , C and f of z . So, what does clause coverage mean? Clause coverage will mean that you make the first clause x greater than y true once and false once. So, I have made this true once because 5, x I have assigned x to be 5, y to be 3 and x greater than y is true and here we have assigned x to be 5, y to be 6, so, x greater than y is false. So, I made the first clause true once and false once, C I have made true once false once, f of z I have made true once and false once. So, this is a set of test cases that achieve clause coverage criteria on a predicate.

So, just to recap predicate coverage says make the predicate true once false once. Predicate coverage need not exercise all the clauses in the predicate. That is not desirable from the point of view of testing because a particular clause that is not exercised for a particular value could contain an error. So, we define another coverage criteria called clause coverage which says you exercise each clause to be true once and to be false once. So, that is what we did for this example.

Now, just for now let us say, let us try to understand what will happen to the predicate for the first assignment of clauses. If you see first assignment of clauses makes all the 3 predicates true. So the entire, I mean, makes all the 3 clauses true. So, the entire predicate will be true second one makes all the 3 clauses false. So, the entire predicate is false. So, in this case clause coverage also happened to achieve predicate coverage, the

set of test cases that we wrote for clause coverage also happened to satisfy predicate coverage.


(Refer Slide Time: 11:48)

Predicate coverage vs. clause coverage

- Clause coverage does not subsume predicate coverage.
- Consider $p = a \vee b$. The clauses are a and b . There are four test inputs that consider all combinations of true/false values for a and b .

	a	b	$a \vee b$
1	T	T	T
2	T	F	T
3	F	T	T
4	F	F	F

- The test set $\{2, 3\}$ above satisfies clause coverage but the predicate is true in both the cases. Hence, predicate coverage is not satisfied.



But that need not be the case in general. Clause coverage in general need not subsume predicate coverage. The notion of subsumption is the same as that we saw for graph coverage criteria. We say one coverage criteria subsumes the other if every set of test cases that satisfy coverage criteria 1 also satisfies coverage criteria 2.

So, the claim now is that clause coverage does not subsume predicate coverage. Why is that so? Here is an example. Consider the predicate p is equal to a or b it has 2 clauses a and b and here is the entire truth table for that predicate what I have done is I have assigned true false values to a and b and in the last column we have tabulated when a or b will become true. This just happens to be directly the truth table for or because we do not have anything else. Now let us say I want to achieve clause coverage for this predicate a or b . So, I make each clause a and clause b true once and false once. So, I chose rows 2 and 3 for that. If you see in row 2 a is made true and in row 3 a is made false, in row 2 b is made false and in row 3 b is made true. So, between rows 2 and 3, I have made a true and false once and I have made b also true and false once.

So, I have satisfied clause coverage by picking up these 2 values for a and b . Now look at the entire predicate a or b . For both row 2 and row 3 the predicate evaluates to true. So, the predicate, I have not given a test case where the predicate has evaluated to false,

right. So, predicate coverage is not satisfied by choose choosing these 2. So, I can say by using this as an example or as a counter example to say that clause coverage does not subsume predicate coverage. In fact, for the same example, if I had chosen rows 1 and 4 which makes a true once false once, b true once false, then I would have achieved predicate coverage. I would have made the whole predicate true once and false once, but the purpose of this example is not to illustrate that. The purpose of this example is to illustrate that it is possible to have combination of values such that clause coverage is satisfied.


But predicate coverage is not met and these 2 values for rows 2 and 3 for the clauses a and b happen to be that. So, in general I cannot say the clause coverage always subsumes predicate coverage, it is a wrong to sayso.

(Refer Slide Time: 14:35)

Combinatorial coverage

- **Combinatorial coverage (CoC):** For each $p \in P$, TR contains test requirements for the clauses in C_p to evaluate to each possible combination of truth values.
- Combinatorial coverage is commonly called as **multiple condition coverage**.
- For the predicate $p = a \vee b$, combinatorial coverage will test all the values in the truth table below.

	a	b	$a \vee b$
1	T	T	T
2	T	F	T
3	F	T	T
4	F	F	F

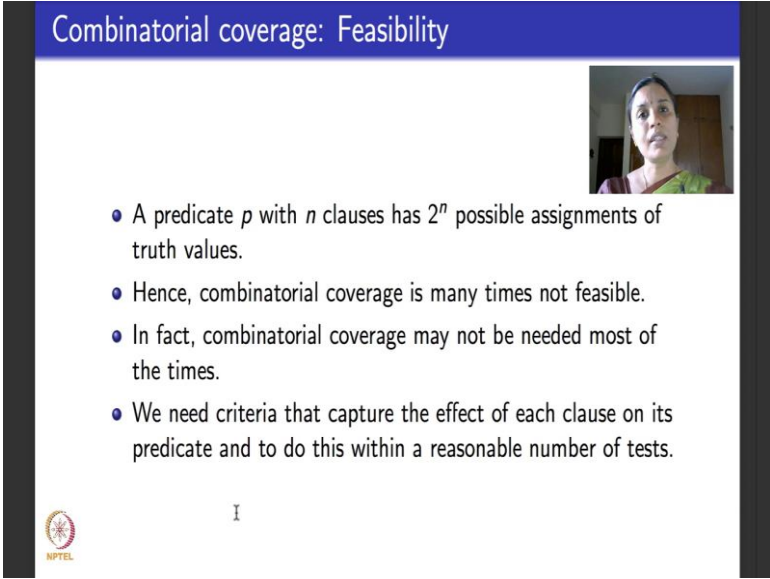


So, the next one would be because there are these combinations where some combination of clauses do not subsume predicates the next one would be to give up and say you test for every possible combination of clause that can occur in the truth table of the given predicate. So, that leads to a coverage criteria called combinatorial coverage abbreviated as CoC.

So, what is combinatorial coverage define? It says for each predicate p in P, TR contains test requirements for the clauses in the C_p to evaluate to each possible combination of truth values. So, if I have, basically what it says is given a predicate p you write out the

truth table for p for each clause in p evaluating to be true or false and then you test for the entire truth table of p . So, if I have p is equal to a or b , there are 4 test cases that I give I met a ; a to be true along with b to be true, then a true b false, a false b true, a false and b false again right, and I test for all possible true false values of the clauses. Now it is not too surprising to see that this might lead to a lot of test cases.

(Refer Slide Time: 15:50)



Combinatorial coverage: Feasibility

- A predicate p with n clauses has 2^n possible assignments of truth values.
- Hence, combinatorial coverage is many times not feasible.
- In fact, combinatorial coverage may not be needed most of the times.
- We need criteria that capture the effect of each clause on its predicate and to do this within a reasonable number of tests.

I

NPTL

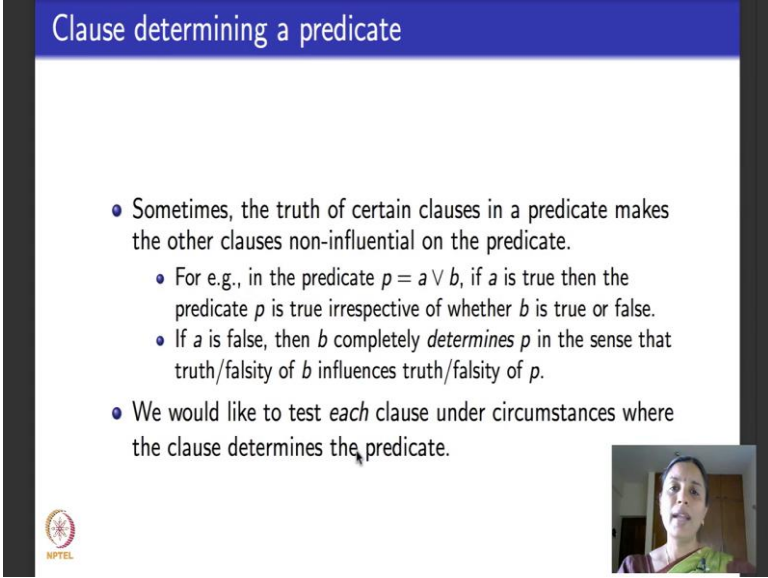
In fact, if I have a predicate with n clauses right and because I am testing against the entire truth table and the truth table has 2 power n rows in the worst case, I will end up getting 2 power n test cases for a predicate with n clauses. That is an exponential number of test cases for a predicate with n clauses. In testing ideally we would like to the test cases to be as minimum in number as possible, but effective in finding faults if they exist. There is no point in testing it for exponential number of combination of clauses and be exhaustive. Of course, this is like exhaustive testing, but it may not be possible to do exhaustive testing all the time and it is not necessary also.

What we are looking for are some kind of criteria that capture the effect of each clause on the predicate and to be able to do this without considering all the combinations of clauses right. Like for example, if you go back to this it so happens that if I choose values for clauses a and b from rows 2 and 3, that combination of values did not really affect the truth or falsity of the predicate. The predicate happened to be true in both the cases. But if I had chosen values 1 and 4 then I was effective in not only achieving clause

coverage, but I was also effective in achieving predicate coverage. I chose different true false values for each of the clauses and as a result I made the entire predicate true once and I made the entire predicate false once. So, that is our ideal goal.

Our ideal goal is to be able to get criteria that capture the effect of each clause on the predicate and to be able to do it without considering an exponential number of combinations. So, the rest of coverage criteria that we will be seeing in this lecture and in part of next lecture, goal is to achieve this particular point.


(Refer Slide Time: 17:48)



Clause determining a predicate

- Sometimes, the truth of certain clauses in a predicate makes the other clauses non-influential on the predicate.
 - For e.g., in the predicate $p = a \vee b$, if a is true then the predicate p is true irrespective of whether b is true or false.
 - If a is false, then b completely *determines* p in the sense that truth/falsity of b influences truth/falsity of p .
- We would like to test *each* clause under circumstances where the clause determines the predicate.

NPTEL

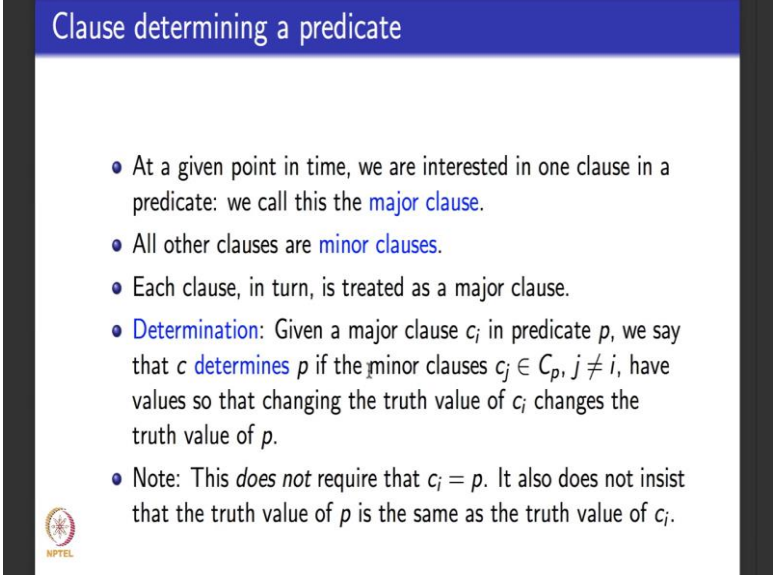


So, for that we need to be able to understand the notion of a clause determining a particular predicate. So, if you take this example p is equal to a or b , let us say a is true right. In this example a happens to be true. If a happens to be true because p is a or b then no matter what the value of b is the entire predicate p will evaluate to be true. Why, because it is an or and there is one for or to be true one of the operands it is enough to be true.

So, we can say that when a is true b does not determine p , which means irrespective of the value that b takes p will become true. But let us say we consider the other case where a happens to be false. If a is false, now whether b is true or not will completely determine whether p is true or not. Let us say a is false, we parked it with that let us say b becomes true then false or true p will become true. Now let us say a being false b becomes false then p will become false. So, if a is false then we say that the clause b completely

determines p in the sense that the truth or falsity of b influences the truth or falsity of the predicate p . So, this is the kind of combination that we would like to test. We would like to test each clause under circumstances where that clause determines the predicate.

(Refer Slide Time: 19:25)



The slide has a blue header with the title "Clause determining a predicate". Below the header, there is a list of five bullet points. The first three points are general statements about clause determination. The fourth point is a definition of "Determination" involving a major clause c_i and minor clauses c_j . The fifth point is a note clarifying that the definition does not require $c_i = p$ or that the truth values of p and c_i must be the same. In the bottom left corner of the slide, there is a small circular logo with the text "NPTEL" below it.

- At a given point in time, we are interested in one clause in a predicate: we call this the **major clause**.
- All other clauses are **minor clauses**.
- Each clause, in turn, is treated as a major clause.
- **Determination:** Given a major clause c_i in predicate p , we say that c_i **determines** p if the minor clauses $c_j \in C_p, j \neq i$, have values so that changing the truth value of c_i changes the truth value of p .
- Note: This *does not* require that $c_i = p$. It also does not insist that the truth value of p is the same as the truth value of c_i .

So, how do we do it ? We have to define what the notion of the clause determining a predicate case first. So, we say at any point in time given a predicate with several clauses I focus my attention on one particular clause. The clause on which I focus my attention on is called the major clause at that point in time. All the other clauses are called minor clauses.

So, if I take for example, this one, the first what I have made is that I say I want to focus my attention on the clause a . So, when I want to focus my attention on the clause a , I make a the major clause. When I make a the major clause, b becomes the minor clause and when I make a the major clause, I want a to determine p . For a to determine p b should be made false. So, at any given point in time we are interested in one clause, we call it the major clause and the rest of the clauses are called minor clauses and then we test how this major clause that we have fixed determines the predicate p .

After finishing this test we pick up another clause from the predicate p , call that other clause that we have picked as the major clause and the remaining clauses including the one we picked up first become the minor clause. So, each clause in turn will be treated as a major clause. So, now, we define the notion of determination. What is determination?

Given a major clause C_i , I fix one clause to be the major clause C_i . I say that C_i determines p if the minor clause is C_j which means all the other clauses are such that changing the value of C_i changes the value of p . So, if we go back to this example, if a is false, if a is false and a is a minor clause and let us say b is a major clause then b completely determines p whenever a is false. So, that is the definition here.

We say a clause that we fix call it the major clause determines the predicate p if the values of the minor clauses are such that changing the value of the major clause changes the value of p . Please note that it does not mean that the major clause is equal to p , neither does it mean that the truth value that the p takes is the same as the truth value of C_i . Let us say the major clause is true, it does not mean that the p predicate p also becomes true exactly when the major clause is true. It could be the case that when the major clause is true the predicate is false and when the major clause is false the predicate is true. Or, it could be the case that when the major clause is true and the predicate is also true and if the major clause is false and the predicate is also false.

But what is important is that I fix one clause, call it the major clause and say this clause has to determine p , means whenever this clause changes its truth value from true or false or false to true, it will change the truth or falsity of p . All the other clauses called minor clauses take values such that this happens and when this happens I say that the major clause determines p .

(Refer Slide Time: 22:53)

Active clause coverage

- **Active Clause Coverage (ACC):** For each $p \in P$ and each major clause $c_i \in C_p$, choose minor clauses $c_j, j \neq i$, so that c_i determines p .
TR has two requirements for each c_i : c_i evaluates to true and c_i evaluates to false.
- For e.g., if $p = a \vee b$, we get four requirements in TR, two for clause a and two for clause b . b needs to be false for a to determine p and vice versa.

	a	b
$c_i = a$	T	F
	F	F
$c_i = b$	F	T
	F	F

- There are three distinct TRs above:
 $\{(a = T, b = F), (a = F, b = T), (a = F, b = F)\}$.

So, using this we define a few other coverage criteria. The first coverage criteria that we will be seeing is what is called active clause coverage abbreviated as ACC. So, what does it say? It says for each predicate p in the set of all predicates P , fix a clause in p , call it the major clause let it be clause C_i and you choose minor clauses C_j 's, all the other clauses such that C_i determines p . Active clause coverage says then there are 2 requirements in my test requirement for active clause coverage. It says one requirement says that the major clause C_i should evaluate to true and the other requirement says that the major clause C_i should evaluate to false. Just to repeat what I told you in a different style, I have a predicate p . The predicate p has several clauses. I fix one clause that I want to focus my attention on let us say clause C_i . I call that the major clause now rest of the clauses are all minor clauses, I make them take values in such a way that this major clause C_i determines p .

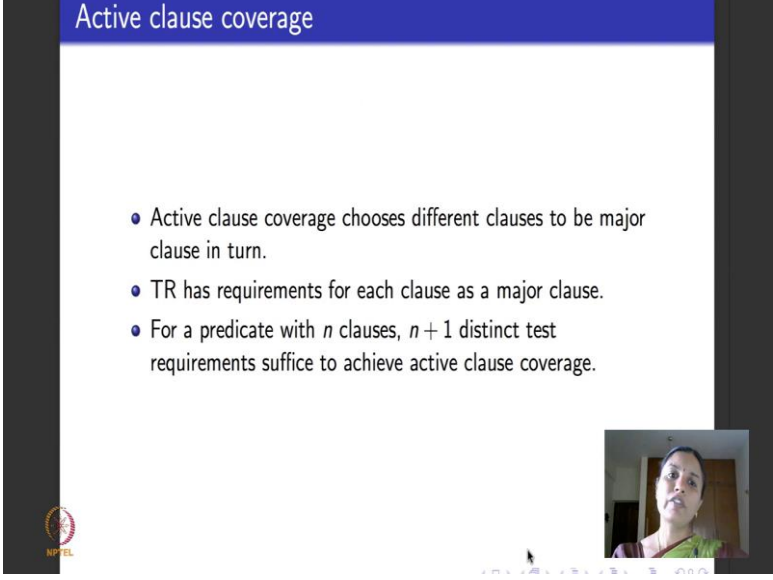
Now, my TR for active coverage criteria says that you make this major clause C_i true once and then you make the major clause C_i false once. Is it clear? So, here is the same example: p is equal to a or b , there are 2 clauses here clause a and clause b . As I told you before we take each clause in turn and make it the major clause. Let us say, to start with I make a the major clause. So, C_i in this table happens to be a , read it as a is the major clause I want a to determine p . So, if a has to determine p then b must become false. So, b is assigned false here in the second column, third column of the table and now my test requirement for active clause coverage says you make a true once, a false once. So, I have given that in bold.

Now, I am done with a being the major clause. It is b 's turn, b 's turn to be the major clause. So, I say C_i is b now for b to determine p a should be false and to achieve active coverage criteria TR, I made b true once and false once right. So, this table just to summarize it, each clause in the predicate takes turns be the major clause. Here there are only 2 clauses a and b . To start with a is the major clause.

So, TR for active clause coverage says make a true once false once. So, I have made a true once false once and I want a to determine p . If I want a to determine p , I have to make b false. Similarly when b is the major clause, a has to be made false and b has to be made true once and false once. If you see these 4 rows, they are not different there is a repeat. Second row and fourth row are repeats of each other. Both in the second row and the fourth row, both a and b are false.

So, I remove it. If I remove it, I only get 3 distinct tests one that makes a true b false which is the first row, the second one makes a false b true which is the third row here and the third one which stands for the combined second and fourth one makes both a and b false.

(Refer Slide Time: 26:37)



The slide is titled "Active clause coverage" in a blue header. It contains three bullet points:

- Active clause coverage chooses different clauses to be major clause in turn.
- TR has requirements for each clause as a major clause.
- For a predicate with n clauses, $n + 1$ distinct test requirements suffice to achieve active clause coverage.

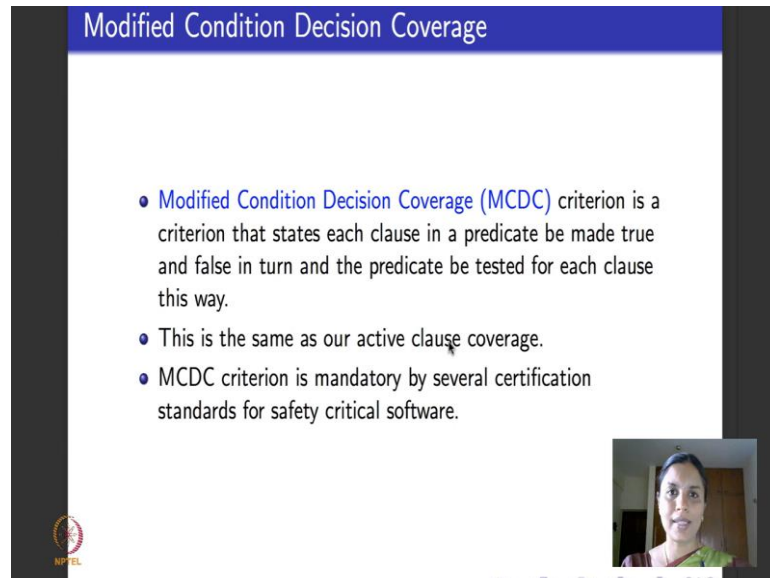
In the bottom right corner, there is a small video inset showing a woman speaking. At the bottom left, there is a small NPTEL logo.

So, what does active clause coverage do? Active clause coverage chooses each clause in the predicate p to be a major clause and then it writes TR, makes the major clause determine the predicate p and writes TR which says now make the major clause true once make the minor clause true once. So, that is the term active clause coverage.

So, let us say a predicate has several clauses. Each clause takes turn to be major and the clause that is major becomes active in the sense that determines the value of the predicate p . So, I make it true once and I make it false once and because I make it true once and I make it false once, it would completely test the predicate p because that determines the predicate p . The predicate also would become true or false once and true or false once again right. It so happens that for a predicate with n clauses active coverage can be achieved by writing n plus one test requirements. Here it looks like we had $2n$ test requirements right, where n is the number of clauses. So, 2 requirements for a, 2 requirements for b, but as this example illustrates one of it was a repeat requirement. So, we merged them and we happened to get 3 requirements. So, in general for a predicate

with n clauses to get active clause coverage it is enough to have n plus one distinct test requirements.

(Refer Slide Time: 28:06)



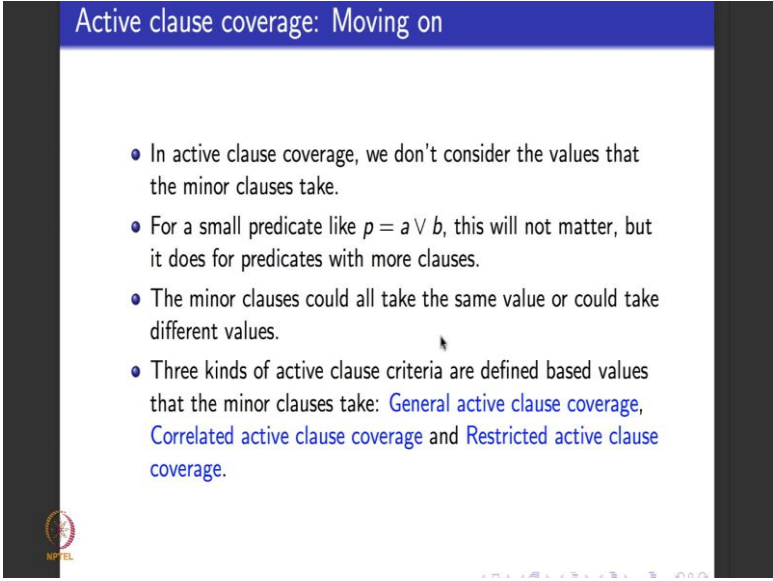
The slide is titled "Modified Condition Decision Coverage" in a blue header. It contains three bullet points:

- **Modified Condition Decision Coverage (MCDC)** criterion is a criterion that states each clause in a predicate be made true and false in turn and the predicate be tested for each clause this way.
- This is the same as our active clause coverage.
- MCDC criterion is mandatory by several certification standards for safety critical software.

In the bottom right corner, there is a small video inset showing a woman speaking. The NPTEL logo is visible in the bottom left corner of the slide.

So, just a small point of observation before we move on. In testing you might have heard of this term called MCDC criteria or MCDC testing. MCDC stands for modified condition decision coverage. It is a criteria which says that when I have a predicate with several clauses, you make each clause in the predicate become true once false once and test the predicate. Usually you switch off all the compiler optimizations, make every clause important in the predicate and exercise each clause to be true once false once and test and see how it influences the predicate. And you can go on modifying this condition what is popularly called as MCDC is what we call as ACC and MCDC is a mandatory way of testing for certifying several safety critical software.

(Refer Slide Time: 28:59)



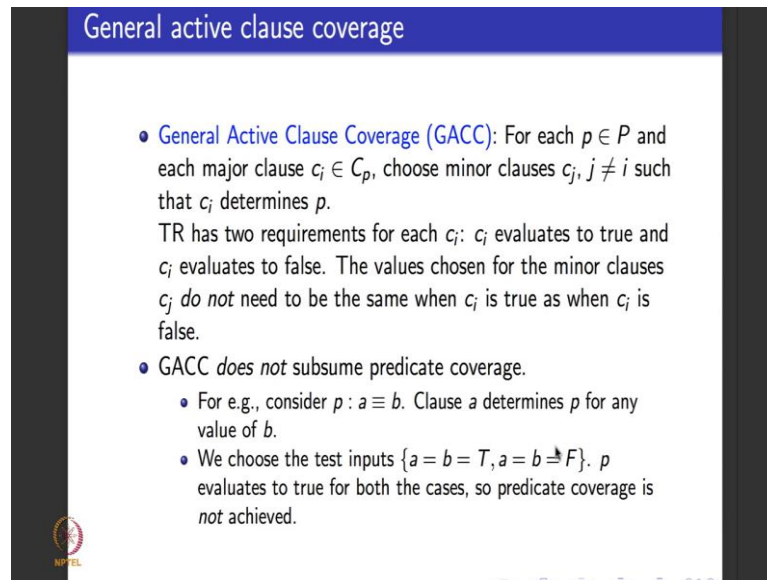
Active clause coverage: Moving on

- In active clause coverage, we don't consider the values that the minor clauses take.
- For a small predicate like $p = a \vee b$, this will not matter, but it does for predicates with more clauses.
- The minor clauses could all take the same value or could take different values.
- Three kinds of active clause criteria are defined based values that the minor clauses take: General active clause coverage, Correlated active clause coverage and Restricted active clause coverage.

NPTEL

So, now we come back to active clause coverage criteria. In active clause coverage criteria we said you check one clause make it the major clause, let the minor clauses take values such that the major clause determines the predicate. We do not really worry about what will be the values that the minor clauses will take right. Will they all take same values, will they all take different values we did not really think about it. If we start thinking about the kind of values that the minor clauses will take, you can further refine active clause coverage criteria into 3 different categories. That is what we are going to do now. We will refine it into 3 different categories: one will be called generalized ACC, the other one will be called correlated ACC and the third one will be called restricted ACC.

(Refer Slide Time: 29:51)



General active clause coverage

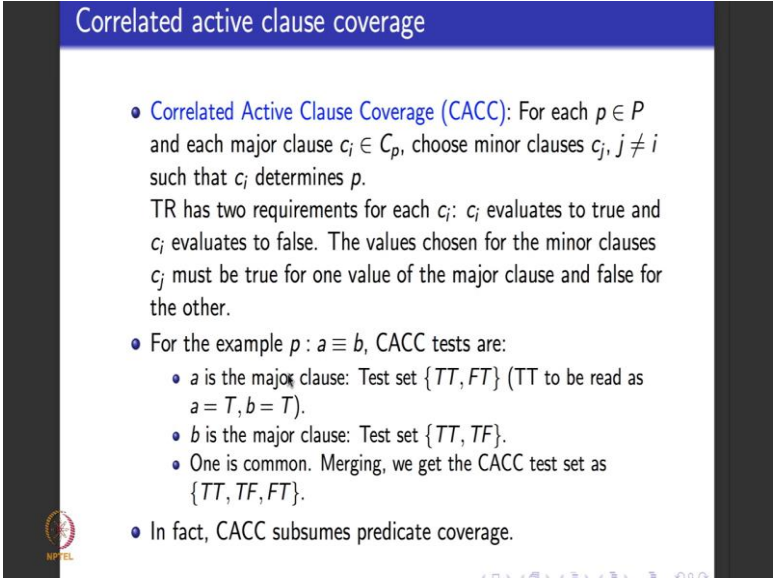
- **General Active Clause Coverage (GACC):** For each $p \in P$ and each major clause $c_i \in C_p$, choose minor clauses $c_j, j \neq i$ such that c_i determines p .
TR has two requirements for each c_i : c_i evaluates to true and c_i evaluates to false. The values chosen for the minor clauses c_j *do not* need to be the same when c_i is true as when c_i is false.
- GACC *does not* subsume predicate coverage.
 - For e.g., consider $p : a \equiv b$. Clause a determines p for any value of b .
 - We choose the test inputs $\{a = b = T, a = b = F\}$. p evaluates to true for both the cases, so predicate coverage is *not* achieved.

So, we will define each one of them one after the other and see what they mean. The first one that we will look at is what is called general active clause criteria GACC. What does it say ? It says the following for each predicate p in P each major clause C_i choose minor clause C_j such that C_i determines p . This is the same active clause coverage I have not changed anything. The next sentence is also the same I have not changed anything TR has 2 requirements for each C_i ; C_i evaluates to true and C_i evaluates to false. Again this is the same as active clause coverage. So far no changes. This is the sentence where we begin to see the change it says the values chosen for minor clauses do not have to be the same as when C_i is true and as when C_i is false. Basically it says do not put any condition on the values that the minor clauses take. Let them be what they are.

So, in some sense generalized active clause coverage is the same as active clause coverage. So, one thing to be noted is that active clause coverage or generalized active clause coverage does not subsume predicate coverage. Why ? Here is an example. Suppose you consider this predicate p which says a is equivalent to b . Now clause a determines p for any value of b right. Suppose b is true then clause a will completely determine p because a could be true or false and change the value of p and suppose b is false then clause a still determines p because a could be true or false and change the value of p right. Suppose we choose the following test inputs. We make a and b both to be true and a and b both to be false right. Then what happens for this predicate p a equivalent to b ? In both the cases the predicate evaluates to true because true is

equivalent to true and false is equivalent to false. So, it again evaluates to true the entire predicate p . But this means that predicate coverage is not achieved right? But I have achieved generalized active clause coverage because I have done a to be the major clause and b also to be the major clause. So, generalized active clause coverage does not subsume predicate coverage.

(Refer Slide Time: 32:23)



Correlated active clause coverage

- **Correlated Active Clause Coverage (CACC):** For each $p \in P$ and each major clause $c_i \in C_p$, choose minor clauses $c_j, j \neq i$ such that c_i determines p .
TR has two requirements for each c_i : c_i evaluates to true and c_i evaluates to false. The values chosen for the minor clauses c_j must be true for one value of the major clause and false for the other.
- For the example $p : a \equiv b$, CACC tests are:
 - a is the major clause: Test set $\{TT, FT\}$ (TT to be read as $a = T, b = T$).
 - b is the major clause: Test set $\{TT, TF\}$.
 - One is common. Merging, we get the CACC test set as $\{TT, TF, FT\}$.
- In fact, CACC subsumes predicate coverage.

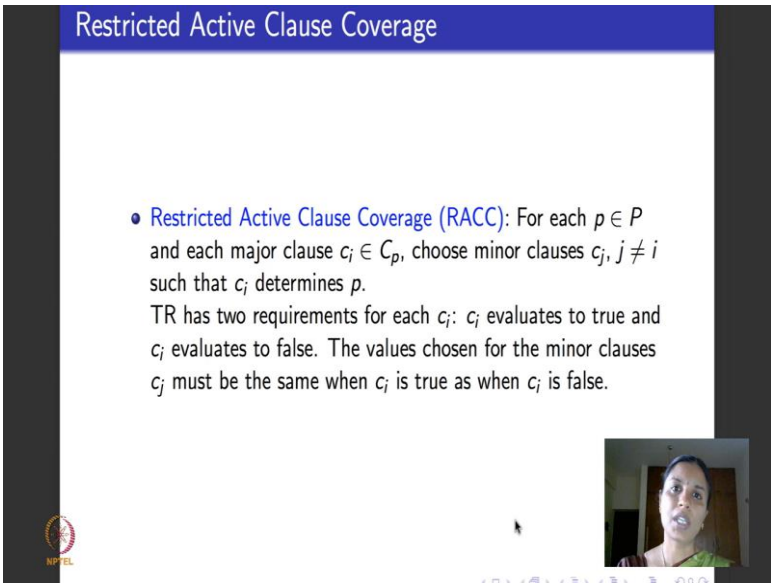
The next condition we will see of ACC criteria is what is called correlated active clause coverage. Now let us focus on the English meaning correlated. What does correlate mean? Correlate means that the minor clauses are such that they correlate with each other. So, the again the first parts of the definition are exactly the same I take a predicate and I fix one clause call it the major clause choose minor clauses such that the major clause determines p no changes so far. Second again no changes the TR has 2 requirements the major clause evaluates to true major clause evaluates to false. The third sentence is again where you begin to see the change. It says the value chosen for minor clauses must be true for one value of major clause and false for one value of the major clause. Go back, what does generalized active coverage criteria say? It says the value chosen for minor clauses need not be the same.

Basically it says do not provide any restriction on the values chosen for minor clauses that is what GACC says. What does CACC say ? It says that the values chosen for the minor clauses must be true for one value of the major clause and false for other value of

the major clause. So we take the same example predicate p which says a is equivalent to b . What are correlated active clause coverage tests for p ? Let's say a is the major clause. Then here is the test set for correlated active clause coverage please read this TT as a is assigning true to a and assigning true to b read FT as assigning false to a and assigning true to b . So, I have made a as my major clause, a is true once false once right. So, I have done this and to make a determine p , I make b true once and true once again. So, this will together determine CACC.

Now, if b is the major clause b becomes true once, false becomes true once and a remains true in both the cases. Here again if you see a is true b is true a is true b is true is common across these things I do not repeat it I put only once. So, I make the 3 tests that put together that satisfy CACC are a true b true, a true b false, a false and b true. In fact, we will show later in the next lecture that CACC does subsume predicate coverage, remember GACC does not subsume predicate coverage and CACC does subsume and I will tell you why in the next lecture.

(Refer Slide Time: 35:15)



Restricted Active Clause Coverage

- **Restricted Active Clause Coverage (RACC):** For each $p \in P$ and each major clause $c_i \in C_p$, choose minor clauses $c_j, j \neq i$ such that c_i determines p . TR has two requirements for each c_i : c_i evaluates to true and c_i evaluates to false. The values chosen for the minor clauses c_j must be the same when c_i is true as when c_i is false.

The last active clause coverage criteria that we will be seeing is what is called restricted active coverage clause criteria. Here again the first few parts of the definition are the same: I take a predicate I take a major clause choose minor clauses such that the major clause determines the predicate. We are here now, TR has 2 requirements for the major clause, major clause evaluates to true major clause evaluates to false. The third line is

where the difference begins to show up. It says here for RACC the value is chosen for minor clauses must be the same as when C_i is true and when C_i is false.


So, we will go back. We saw active clause coverage criteria. We are refining active clause coverage criteria into 3 kinds: general active clause coverage criteria, correlated active clause coverage criteria and restricted active clause coverage criteria. What does general say? General says that the major values chosen for the minor values do not have to be the same there is no condition that is why it is called generalized. What does correlated say? Correlated says that the values chosen for minor clauses must be true for one and false for the other that is why the term correlated. What does restricted say? Restricted says that the values for the minor clauses must be the same as when the major clause is true as and when the major clause is false. So, it is restricted because it restricts the truth or falsity of the minor clauses to be the same as that of the major clause.

(Refer Slide Time: 37:02)

ACC Criteria: Example

- We illustrate the various ACC criteria using another example.
- Consider the predicate $p = a \wedge (b \vee c)$.
- Truth table for p .

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F



So, since this is a little complicate, what I thought I will do is I will take another example and explain active clause coverage criteria with reference to this example to help you understand it well. So, here is the second example. Now you consider this predicate, read it as p is equal to a and b or c . What I have done here is I have given you the entire truth table for the predicate. So, there are 3 clauses in this predicate: a , b and c . Each of these 3 clauses can take 2 false values in combinations. So, if I put 2 values per clause then I totally get eight combinations of true false values as listed here and the final column in

this truth table says what will happen for each of the combinations of true false values of a, b and c. For example, if all 3 are true which is the first row: a, b and c are true then what will happen to a and b or c, it will be true. Why, because b or c will be true and true and true will be true.

Now, you take some third row here it says a is true b is false c is true, but the result the whole predicate becomes true. Why is that, because even though b is false; false or true b or c will evaluate to be true that ended with a being true will give me true. Let us take another example let us take row number six. It assigns a to be false, b to be true, c to be false and because a is false and a is ended with something the whole predicate becomes false. So, this table gives the entire truth table for this predicate. Now what we are going to do is we are going to understand how the various active clause coverage criteria work for this predicate. We begin, as I told you generalized active clause coverage criteria is the same as ACC nothing interesting.



(Refer Slide Time: 38:50)

CACC on $p = a \wedge (b \vee c)$

- In $p = a \wedge (b \vee c)$, for a to determine p, $b \vee c$ must be true.
- $b \vee c$ can be made true in three ways.
- CACC is satisfied for a being the major clause by choosing one TR from rows 1,2,3 and one from rows 5,6,7.

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F

- Totally, nine combinations exist.

So, we will see correlated active class coverage criteria for these predicate and restricted active class coverage criteria for the predicate. So, let us start with CACC, correlated active clause coverage criteria. So, now I want to make a the major clause which means what I want a to determine the predicate p. For a to determine the predicate p this must be true right because if b or c is true then if a is true p will become true and if a is false p will become false. Now you look at b or c, b or c can be made true in 3 ways right. You

could make both b and c true, you could make one of b or c true. So, in 3 different possible combinations of true false values for b and c, the predicate b or c will be true.

So, what I have done is I have pulled out rows from this truth table. Only those rows which will help you to get the coverage criteria of CACC satisfied. So, CACC for a being the major clause can be satisfied by making a true once, which means choosing one of rows to one 2 and 3. Please see here, all the 3 rows a is true right and in all the 3 rows there are 3 possible ways of making b or c true. So, a and b or c becomes true. In rows 5, 6 and 7, a is false a is the major clause I made it true in rows 1 2 and 3 now I am making it false in rows 5, 6 and 7. Now again b or c is true 3 different possibilities in all the 3 cases the predicate becomes false. So, to get my test cases for CACC, I pick up one from the top set one 2 3 and one set of values from the bottom set 5, 6, 7. Any of these will make a true once false once and b or c combinations are such that a will determine p and they also satisfy correlated active clause condition right. It says the values chosen for minor clauses must be true for one value for the major clause and false for the other that is also satisfied.



So, to satisfy CACC for the predicate a and b or c, pick up one row from here one row from here. How many different ways of picking up one row from here one row from here exist? 3 into 3 nine ways. So, nine combinations are of nine different test cases can be written to satisfy CACC for this predicate.

(Refer Slide Time: 41:41)

RACC on $p = a \wedge (b \vee c)$

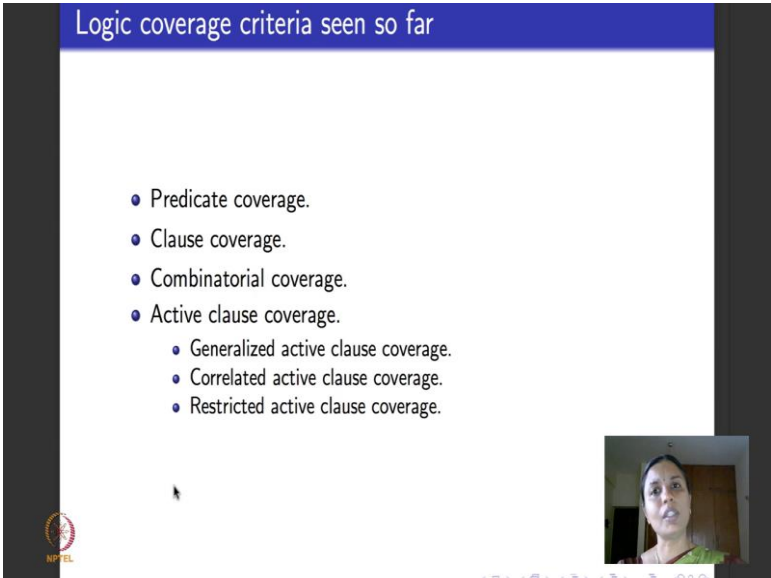
- Clause a is the major clause: Only three of the nine sets of test requirements that satisfy will satisfy RACC.
- Row 1 is paired with row 5, row 2 is paired with row 6 and row 3 is paired with row 7.

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
5	F	T	T	F
2	T	T	F	T
6	F	T	F	F
3	T	F	T	T
7	F	F	T	F

Now for the same predicate if you take a again as the major clause and instead of doing CACC you try to do RACC restricted active clause coverage. In which case what you can do is a is the major clause. So, I make a true once false once, true once false once, true once false once and the values of minor clauses should be such that they take the same value right you remember that they take the same value that is what restricted active class coverage says. So, here if you see b is true and c is also true b is true, c is also true. In which case a determines the whole predicate p the whole predicate becomes true once false once. Similarly here b and c take the same value such that a determines p. Again b and c take the same value such that a determines p. So, here is how I satisfy RACC. So, I any of these 3 combinations 1 with 5, 2 with 6 and 3 with 7 will satisfy restrictive active clause coverage criteria on this predicate.

(Refer Slide Time: 42:50)



Logic coverage criteria seen so far

- Predicate coverage.
- Clause coverage.
- Combinatorial coverage.
- Active clause coverage.
 - Generalized active clause coverage.
 - Correlated active clause coverage.
 - Restricted active clause coverage.

So, what are the logic coverage criteria that we have seen? We started with predicate coverage, most elementary coverage criteria make the whole predicate true once false once. Then we moved on to clause coverage which says for each clause in the predicate make it true once false once. We realize that they do not subsume each other, they can be completely unrelated we saw examples for that. Then we say why bother, you take every combination of true false values of the clauses in the predicate. That will give rise to combinatorial coverage. Then we realized that there were too many combinations. There were in fact, exponential number of combinations it is not worth it.

So, I say I want to look at active class coverage where at any point in time I have focused on one clause in the predicate call it major clause all the other clauses are minor clause and see how these major clause determines p. And there are 3 different ways in which the minor clauses could take values, those give rise to 3 different active class coverage criteria. In the next lecture we will move on and define some more coverage criteria and I will tell you a little bit about how to actually get test cases to satisfy these coverage criteria.


Thank you.

Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 23
Logics: Coverage Criteria, contd

Hello everyone. We are in the middle of week 5. What I will be doing today is the 4th lecture of week 5. We did assignments solving for the last week then we began with logic coverage criteria, I introduced to you the basics of logic in the second lecture of this week. Then last lecture we started seeing coverage criteria. Today we will continue to see coverage criteria.

(Refer Slide Time: 00:42)



Recap: Logic coverage criteria seen so far

- Predicate coverage.
- Clause coverage.
- Combinatorial coverage.
- Active clause coverage.
 - Generalized active clause coverage.
 - Correlated active clause coverage.
 - Restricted active clause coverage.

The slide is a presentation slide with a blue header and a white body. It lists five types of logic coverage criteria. The first four are Predicate coverage, Clause coverage, Combinatorial coverage, and Active clause coverage. The fifth, Active clause coverage, has three sub-points: Generalized active clause coverage, Correlated active clause coverage, and Restricted active clause coverage. There is a small NPTEL logo in the bottom left corner and a video feed of the professor in the bottom right corner.

We will finish doing logic coverage criteria. In the next class, I will tell you about algorithms for logic coverage criteria. So, what were the coverage criteria that we have seen till now? We saw what were predicates and clauses. Predicate is like any expression that evaluates to true or false that labels decision statements that you find in programs and guards that you could find in design or requirements.

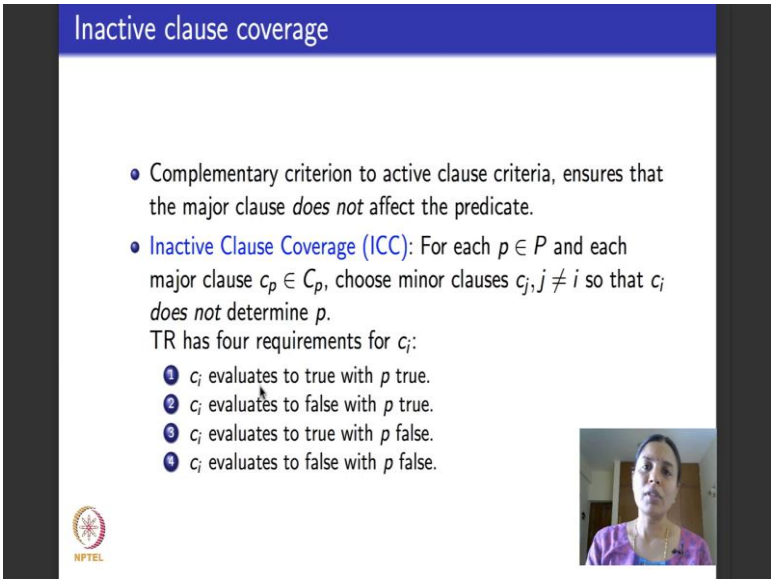
So, the predicate has one or more clauses which are atomic entities in predicates. Clauses do not have logical connectors. So, we saw predicate coverage which tested the predicate to be true or false, clause coverage which tested each clause in the predicate to be true or false in turn. Then we saw what was called all combinations coverage, sometimes it is

also called complete clause coverage where they test the entire predicate for every possible combination of true false values in clauses. This will result in exponential number of test cases, usually not desirable.

So, what we try to want to test is how each clause affects a predicate in turn and, which are the important clauses, which will affect the predicate in the sense that the truth or falsity of the clause clearly determines the truth or falsity of the predicate. So that gave rise to what we call as active clause coverage, and we realize that active clause coverage did not really tell you what the other clauses would do. We fix the clause called the major clause see how the major clause determines the predicate, but we did not really specify how the other clauses put together called as minor clauses affect the truth or falsity of the predicate. What we did was we went ahead and refined active clause coverage into 3 specific coverage criteria called generalized active class coverage, correlated active clause coverage and restricted active clause coverage.



So, in generalized, we do not have any conditions that tell you what are the values that the minor clauses take. In correlated, we say that the minor clauses take values such that they aid the major clause to completely determine the predicate and in restricted, we say the minor clauses all have to take the same value as the predicate is true and as and when the predicate is false.

(Refer Slide Time: 02:52)



Inactive clause coverage

- Complementary criterion to active clause criteria, ensures that the major clause *does not* affect the predicate.
- **Inactive Clause Coverage (ICC):** For each $p \in P$ and each major clause $c_p \in C_p$, choose minor clauses $c_j, j \neq i$ so that c_i *does not* determine p .
TR has four requirements for c_i :
 - 1 c_i evaluates to true with p true.
 - 2 c_i evaluates to false with p true.
 - 3 c_i evaluates to true with p false.
 - 4 c_i evaluates to false with p false.

Now, moving on we will today look at what is called inactive clause coverage criteria. So, here again we want to know, we want to know per clause in a predicate how that clause influences the predicate in active clause coverage criteria. In inactive clause coverage criteria, we fix a clause again call it the major clause, but our interest is knowing how the major clause does not affect the predicate. This might be a little counter intuitive. Very soon I will tell you an example that motivates for the need for a clause criteria like inactive clause criteria. So, what is the basis with which we start inactive clause coverage criteria? So, there is a predicate, there are several clauses in the predicate, each clause in the predicate takes turn to become what is called the major clause.

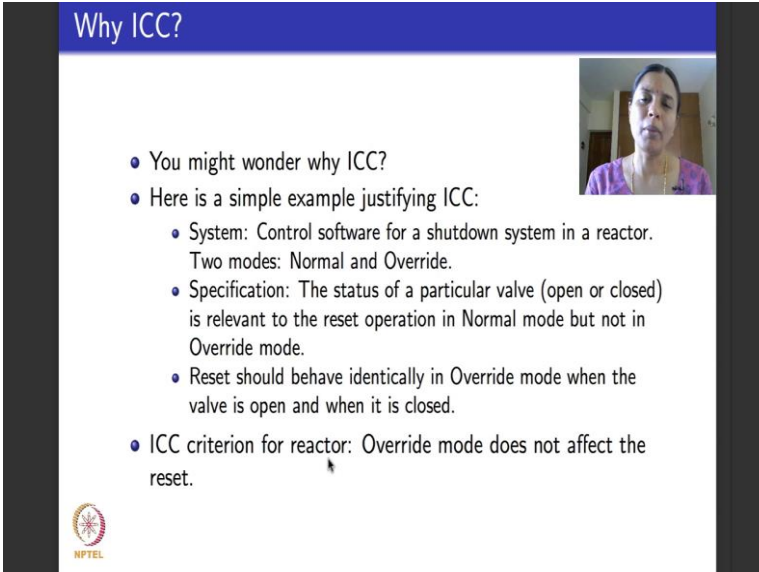
So, let us fix one clause in the predicate, call it a major clause. I want the other clauses minor clauses to take values in such a way that the major clause does not affect the predicate. So, that is what is called inactive clause coverage criteria. So, what is inactive clause coverage abbreviated as ICC? It says for every predicate p in the set of predicates P , each clause takes turns to be a major clause let us say c_i is a major clause we chose minor clause as c_j such that c_i does not determine p . The test requirement has 4 requirements for c_i : the predicate p can evaluate to be true in the first 2 cases in which case the c_i major clause evaluates to true once and to false once. In the second case the predicate p evaluates to false and the major clause evaluates to true and evaluates to false.

So, put together if you read all these 4 TRs, it says the predicate p can evaluate to true once and predicate p evaluates to false once and for each of these combinations of the predicate p evaluating to true and false, the major clause c_i evaluates to true once and false once each time. So, the major clause has no influence on p . In other words the major clause does not determine p . So, we say the major clause is inactive as far as the truth or falsity of p is concerned and this is what is called inactive clause coverage. Another thing that I would like you all to observe before we move on is look at these 4 test requirements: it says p evaluates to true in the first two p evaluates to false in the first two.

So, in both the cases between both the cases predicate coverage is true. So, inactive clause coverage criteria subsume predicate coverage. So, towards the end of this module we will see logic coverage criteria subsumption. There we will tell you what happens


right. So, just to repeat and make sure you understand what inactive clause coverage is. So, I fix one clause, call it the major clause and my goal is to make sure that the major clause has no influence on the truth or falsity of p . In other words the major clause does not determine the predicate p . So, my TR under inactive clause coverage has four requirements: p can become true in the first two, p can become false in the last two. Whenever p becomes true the major clause becomes true once and false once for p becoming false the major clause becomes true once and false once.

(Refer Slide Time: 06:30)



Why ICC?

- You might wonder why ICC?
- Here is a simple example justifying ICC:
 - System: Control software for a shutdown system in a reactor. Two modes: Normal and Override.
 - Specification: The status of a particular valve (open or closed) is relevant to the reset operation in Normal mode but not in Override mode.
 - Reset should behave identically in Override mode when the valve is open and when it is closed.
- ICC criterion for reactor: Override mode does not affect the reset.



So, the major clause that I fix has no influence on the value of p . So, as I told you might wonder why I want to have inactive clause coverage criteria. Why do I want a major clause such that it does not determine p ? It so happens that it is very useful when I consider practical applications of logic coverage criteria. So, here is a simple example that motivates and justifies the need for inactive coverage clause criteria. So, let us take an example system. The example system that we consider is a shut down system of a reactor. So, there is a particular reactor, it could be any kind of reactor let us say nuclear reactor or any other reactor and sometimes I may want to shut down the working of the reactor. How is the shutdown done? Shutdown is done with the help of a software that controls the working and shutting down of a reactor.

So, that software we call it control software right. The control software typically operates in 2 modes or in 2 states. So, it operates in normal mode or it operates in what is called in

override mode. You can interpret normal mode to be normal operations of the control software as it is working with control reactions on the reactor. Override mode is control software is trying to take over may be there is an emergency or something like that, it is trying to take over and do certain actions on the reactor system. Now what are specifications of such a control software. There could be several specifications, but here is a small example that will motivate why we need inactive clause coverage criteria.

So, specification says that now, remember we are going to shut down we are looking at control software that shuts down the nuclear reactor. Let us say shutdown happens through a particular valve. The valve could be open when you want to you want the nuclear reactor to work normally and the valve needs to be closed when you want to shut down the nuclear reactor.

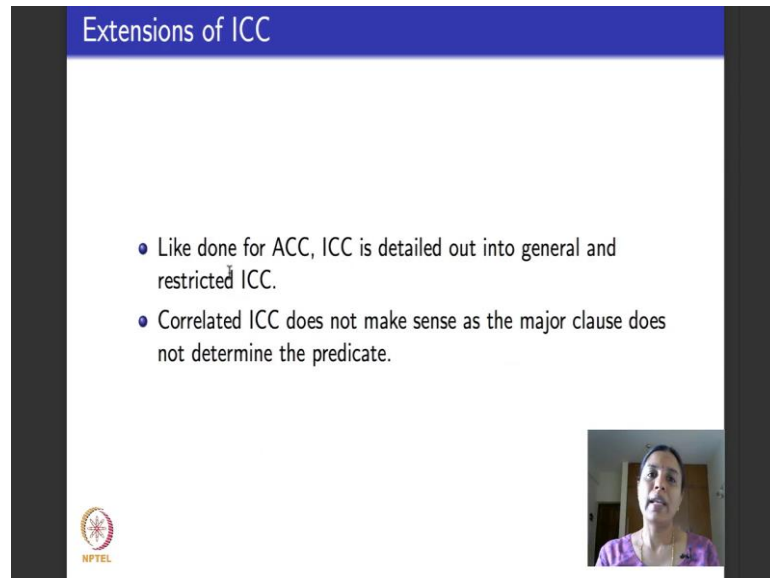
So, the control software actually sends activates the command to open or close this valve. So, specification could be the following it says that a particular valve which could be open or closed is relevant, the status of a particular valve, is relevant to the reset operation in normal mode, but not in override mode. So, what it says is that this reactor could be in normal mode or override mode. So, the shutdown operation, which means opening or closing the valve the status of the valve whether it was already open or already closed, what do I want to do with it, that is relevant to the reset operation in normal mode, but when the control software is working with the reactor in override mode irrespective of what the status of the valve is it will go ahead and shut down the reactor.

So, what it says is that the reset operation, if I consider it as an action, that should behave identically in override mode when the valve is open and when the valve is closed. In other words, stating it in terms of logical coverage criteria, if you think of this as a predicate, it says the clause which says reset is true or false should not influence the predicate, the mode of the predicate. So, it should be the same as an override mode when the valve is open and when the valve is closed. So, it says the particular operation should not influence the state of a valve when the nuclear reactor is in override mode.

So, the particular operation should be inactive if it is modeled as a clause when the reactor is in override mode, inactive to the status of the valve. So, this is a natural case

where you say that if this is modeled as a clause, if it is a major clause it should never influence what happens to the status of the valve.

(Refer Slide Time: 10:16)



The slide is titled "Extensions of ICC" in a blue header. It contains two bullet points: "• Like done for ACC, ICC is detailed out into general and restricted ICC." and "• Correlated ICC does not make sense as the major clause does not determine the predicate." In the bottom right corner, there is a small video inset showing a woman speaking. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.


So, here is the good reason to consider things like inactive clause coverage criteria. Like we did for active clause coverage, if you go back to the first slide where we recapped what we did in the last class we looked at active clause coverage we looked at 3 specific ways of working with active clause coverage. One was general active clause coverage, correlated active clause coverage and restricted active clause coverage. Today we went ahead and defined inactive clause coverage. So, like we did for general active clause coverage you could define extends inactive clause coverage into various specific things.

So, what are the 3 extensions possible? Like we did for active clause coverage you could have generalized correlated and inactive. But if you pause and think for a minute correlated inactive clause coverage criteria does not make sense. Why does it not make sense, because in inactive coverage clause criteria, we say that the major clause does not determine the truth or falsity of the predicate. So, when it does not determine the truth or the falsity of the predicate the minor clauses cannot correlate with each other and assume values that aid in the major clause determining the predicate. So, we do not define correlated inactive clause coverage criteria. It does not make sense. We define only restricted ICC and generalized ICC.

(Refer Slide Time: 11:34)

General Inactive Clause Coverage

- **General Inactive Clause Coverage (GICC):** For each $p \in P$ and each major clause $c_p \in C_p$, choose minor clauses $c_j, j \neq i$ so that c_i *does not* determine p .
TR has four requirements for c_j :
 - 1 c_j evaluates to true with p true.
 - 2 c_j evaluates to false with p true.
 - 3 c_j evaluates to true with p false.
 - 4 c_j evaluates to false with p false.The values chosen for the minor clauses c_j may vary amongst the four cases.





So, here is how generalized ICC is defined. Like we did for active clause coverage criteria you retain the main part of the definition. You retain the main part of the definition of inactive clause coverage criteria then you add a few extra condition which define what the specific extension is. So, in the first part of generalized inactive clause coverage criteria we repeat the definition of ICC that we gave in this slide right. So, I pick up a clause, call that the major clause and choose minor clauses such that major clause does not determine p . TR has four requirements, p evaluates to true and false with major clauses evaluating to true and false in each of the cases.

So, that part is fully repeated, there are no changes. Now what are the extra conditions that we add ? We say that the values chosen for the minor clause is c_j can vary in the 4 cases. So, it is as general as possible, exactly like we defined for active clause coverage. So, the minor clauses values have no conditions, no restrictions associated with it they can vary amongst the 4 cases.

(Refer Slide Time: 12:48)

Restricted Inactive Clause Coverage

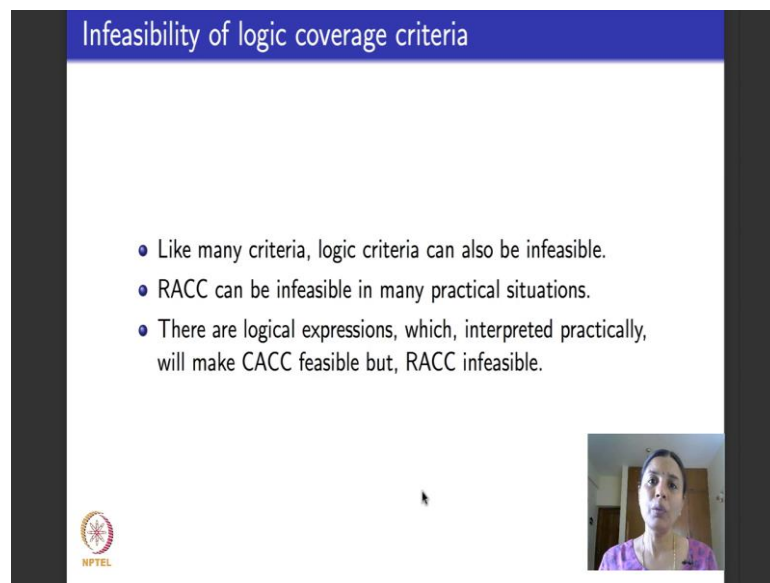
- **Restricted Inactive Clause Coverage (RICC):** For each $p \in P$ and each major clause $c_p \in C_p$, choose minor clauses $c_j, j \neq i$ so that c_j does not determine p .
TR has four requirements for c_j :
 - 1 c_j evaluates to true with p true.
 - 2 c_j evaluates to false with p true.
 - 3 c_j evaluates to true with p false.
 - 4 c_j evaluates to false with p false.The values chosen for the minor clauses c_j must be the same in cases (1) and (2), and the values chosen for the minor clauses c_j must be the same in cases (3) and (4).



Now what is restricted inactive clause coverage criteria ? Again you repeat the entire definition of inactive clause coverage first. So, this part is the same. So, you choose a major clause, choose minor clauses such that the major clause does not determine p . The 4 requirements in TR therefore, p evaluating to true the major clause evaluates to true once false once, for p evaluating to false the major clause evaluates to true once and false once. Now come the extra conditions here below. What are the extra conditions, what do they say? It says the following it says that the value chosen for the minor clauses c_j must be the same for cases one and two. What are cases 1 and 2 ? They correspond to, if you go up and look, they correspond to predicate being true. So, the minor clauses should take the same values when the predicate is true and when the predicate is false, which corresponds to cases 3 and 4, the minor clauses should again take the same value.

So, that is why it is a restricted. It says there is 2 TR s for the predicate becoming true: major clause does not determine the predicate the minor clauses all assume the same value. Now, there are 2 more TRs for the predicate becoming false major clause again does not determine the predicate, minor clauses again take the same set of values. So, this is restricted inactive clause coverage. As I told you we do not define correlated inactive clause coverage because it does not make sense as the major clause does not determine the predicate.

(Refer Slide Time: 14:18)



The slide is titled "Infeasibility of logic coverage criteria" in a blue header. It contains three bullet points:

- Like many criteria, logic criteria can also be infeasible.
- RACC can be infeasible in many practical situations.
- There are logical expressions, which, interpreted practically, will make CACC feasible but, RACC infeasible.

In the bottom left corner is the NPTEL logo. In the bottom right corner is a small video inset showing a woman speaking.


So, we move on. Like we did for graph coverage criteria, it is the case that logic coverage criteria also suffer from several of these TRs corresponding to the various coverage criteria being infeasible. Infeasible means there is practically impossible to write a set of test cases that will actually satisfy the test requirement. So, what do we do? If that is the case then what we do is we simply ignore the coverage criteria or we look at other coverage criteria. I will tell you through an example how infeasibility comes when it comes to logical coverage criteria and how to deal with infeasibility.

It so happens that in many cases restricted active clause coverage criteria or restricted inactive clause coverage criteria turn out to be infeasible. In which case we resort to correlated or general active or inactive clause coverage criteria.

(Refer Slide Time: 15:14)

Feasibility: CASS vs. RACC, through an example

- System: A particular valve can be open or closed, system can be in two modes: "Operational" and "Standby".
- Two constraints on the system:
 - The valve must be open in "Operational" mode and closed in all the other modes.
 - The mode cannot be both "Operational" and "Standby" at the same time.
- Need to test for a certain action being taken only if the valve is closed and the system status is either "Operational" or "Standby".



So, here is another example very similar to the nuclear example that we saw, nuclear reactor example that we saw few slides ago. So, here again there is a system, some system may be a reactor we do not know what it is, it is not needed for, it is controlled by software and then there is a particular valve that this software tries to control through an actuator. That valve as always can be open or can be closed and the system can be in 2 modes. The system can be operational or working and the system can be in standby mode. There are 2 constraints on the requirements of the system, 2 design constraints on the requirements of the system that are listed here. It says that the valve must be open when the system is in operational mode and the valve must be closed in all other modes.

In particular the valve must be closed when system is in standby mode the second constraint says that system cannot be both operational and standby mode that makes sense right you might wonder it makes obvious sense why would I want to write it I would want to write it because I have to be able to be complete while writing my constraints.

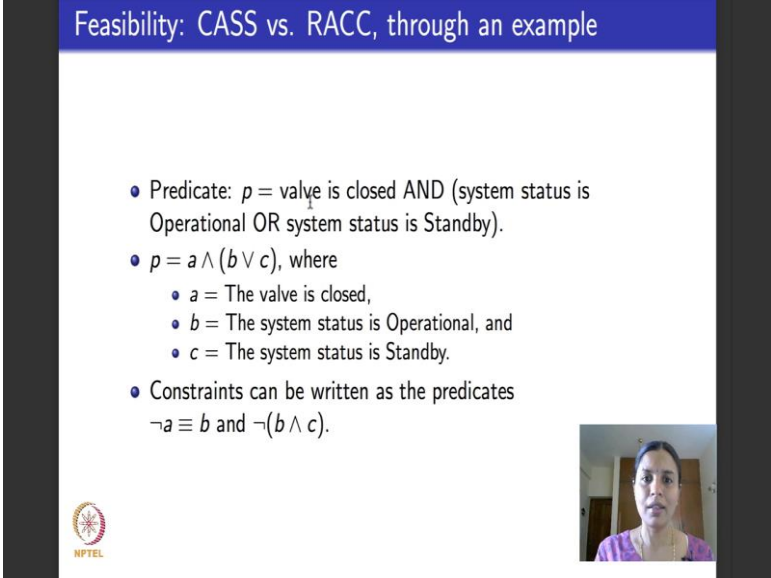
So, just to repeat what is the system? It is some system being controlled by software the software controls the system by opening and closing a valve. There are 2 modes or states in which the system can be in: normal mode, operational mode also called as operational mode and standby mode. The valve can be open or closed. There are 2 constraints on the requirements and design of the system. The first constraint says that the valve must be

open when the system is operational and closed when the system is in standby and it says the system can never be both in the operational state and in the standby state at the same time.

Now, what we want to do is let us say this is our requirement that is given. Somebody comes and tells you test for a certain action being taken only if the valve is closed and the system is either operational or standby. There is some requirement which tells you test this system for the following requirement: the valve is closed and the system status is either operational or standby. So, I take. So, there are 3 constraints here the 2 constraints come as these 2 that are listed here and this last part which gives me a test requirement written in English becomes the main predicate that I have to specify in logic and may be try to use logic coverage criteria to be able to test it.

So, let us read out the predicate once again it says test for what, test for a certain action being taken only if this condition is met. What is the condition? The valve is closed and the system status is either operational or standby.


(Refer Slide Time: 18:05)



Feasibility: CASS vs. RACC, through an example

- Predicate: $p = \text{valve is closed AND (system status is Operational OR system status is Standby)}$.
- $p = a \wedge (b \vee c)$, where
 - $a = \text{The valve is closed,}$
 - $b = \text{The system status is Operational, and}$
 - $c = \text{The system status is Standby.}$
- Constraints can be written as the predicates $\neg a \equiv b$ and $\neg(b \wedge c)$.

NPTL



So, I write it again the valve is closed and the system status is operational or system status is standby. That is the predicate p that I want to focus that I want to be able to test on. So, my p is valve is closed written as ' a ', system status is operational written as ' b ', system status is standby written as ' c '. So, what is the predicate if I translate; if I substitute back for a , b and c in the formula then what is the predicate that I get I will get

the following. I will say $a \wedge b \vee c$, means the valve was closed and the system is either operational or the system is in standby mode.

If you remember this predicate $a \wedge b \vee c$ is precisely the predicate that we looked at in the last lecture when we worked with examples of combinational coverage criteria, generalized active coverage criteria and restricted active coverage criteria. I had given you the truth table for this predicate and told you which rows to choose from and do what in terms of meeting the TRs for the various RACC and CACC coverage criteria.

Now, this is the predicate go back to this example it also had 2 constraints. What were the constraints? The first constraint said that the valve must be open in operational mode and closed in all other modes. So, the valve is closed is modeled as 'a' for me the system status is operational is modeled as 'b'. So, how will I write the first constraint? I will write the first constraint using this formula it reads as $\neg a \supset b$, not a is equivalent to b. What does 'not a' mean? $\neg a$ means the negation of 'a', 'a' says the valve is closed. So, the valve is not closed which means the valve is open. So, it says the valve is open if and only if the system is operational. That is what is the first constraint right? Second constraint says the mode cannot be operational and standby at the same time.

So, there is a clause that we have already designated calling b for the system status being operational and there is another clause, call it c, for the system status being standby. So, I write it like this I say it is not the case that b and c holds together. It is not the case that the system is operational and the system status is standby together $\neg(b \wedge c)$. So, what have I done? My goal is what my goal is to illustrate the difference between CACC and RACC through an example.

So, what have I done? I have taken this example written 2 constraints on this example and the predicate. So, in this slide we have expressed the predicate as a logical formula and we have expressed the constraints as these extra conditions that the predicate, any test requirement on the predicate needs to satisfy before achieving a coverage criteria on the predicate.



(Refer Slide Time: 21:03)

Feasibility: CASS vs. RACC, through an example

Constraints limit the feasible values in the truth table.

	a	b	c	$a \wedge (b \vee c)$	
1	T	T	T	T	violates constraints 1 and 2
2	T	T	F	T	violates constraint 1
3	T	F	T	T	
4	T	F	F	F	
5	F	T	T	F	violates constraint 2
6	F	T	F	F	
7	F	F	T	F	violates constraint 1
8	F	F	F	F	violates constraint 1

Only feasible rows are 3, 4 and 6.



So, let us go back and look at this predicate. If you remember this is the truth table for the predicate that we had written in the last lecture. I have re-copied the entire truth table here in the left hand side and I have added one more column here. What does this extra column say? This extra column populates details about if this was the assignment of truth values of the clauses a, b and c and let us say I am considering this as a potential test case value then can I consider it or can I not? I can consider it provided it does not violate any of the given constraints. So, for every combination of truth values for a, b and c, in the last column, I have taken the truth table and added the last column and I have documented about whether the particular combination of a values for a, b and c violates any of the constraints. So, for the first row, a comment is written here saying that the value of a, b and c being true violates both the constraints one and two. Why is that so?

Let us go back to the previous slide look at the constraints. So, first constraint says $\neg(a \Xi b)$. So, $\neg(a \Xi b)$ cannot be true if both 'a' and 'b' are true right, because $\neg a$ may be false and 'b' will be true. So, that is violated. Second constraint says $\neg(b \wedge c)$. So, $\neg(b \wedge c)$ will be not of true and true, which is not of true which is false right. So, this assignment of truth values for a, b and c all of them being true violates both the constraints. The second assignment where a is true, b is true and c is false violates constraint one.

So, again it violates $\neg(a \Xi b)$ because both a and b are true. So, negation of a cannot be equivalent to b. So, I have does not violate constraint 2 that is fine, but it violates

constraint one. Third assignment of truth values for a, b and c seems to be fine. So, nothing is written about it violating any of the constraint, fourth assignment is also fine, fifth violates constraint 2 because b and c are true together sixth is fine no violation seventh and eighth both violate constraint one. So, if I consider a test requirement for this entire predicate p, the test requirement basically should not violate the additional constraints imposed on the system. So, totally eight possible test requirements are there because that is the exhaustive combinations of true false values of a and b.

Now, it so happens that 5 of them violate some condition or the other one of them violates both the constraints. So, out of the various test requirements, test values I have for Trs, only 3 combinations of true, false values for a and b can be used as test cases because only those 3 of them do not violate any of the constraints. So, those 3 are given by rows 3, 4 and 6.

(Refer Slide Time: 24:07)

Recap: CACC on $p = a \wedge (b \vee c)$

- In $p = a \wedge (b \vee c)$, for a to determine p, $b \vee c$ must be true.
- $b \vee c$ can be made true in three ways.
- CACC is satisfied for a being the major clause by choosing one TR from rows 1,2,3 and one from rows 5,6,7.

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F

• CACC is feasible for the predicate p.

Now let us go back and recap from the previous lecture, what was CACC on this predicate, CACC TR and test cases. The predicate is the same. This is what we saw in the last lecture, we had taken the entire truth table and worked out what this restricted table represents as test cases for CACC. If you remember we said any of these combination of test cases, one for the predicate evaluating to true and one for the predicate evaluating to false will be to satisfy CACC requirements. So, you could pick

up either one anyone from rows 1, 2 and 3 or any one from rows 5, 6 and 6; 5, 6 and 7 sorry and that will be a test case combination that will satisfy CACC.

So, now I take this predicate. My goal is to be able to do CACC for this predicate in the presence of these 2 additional constraints. So, these 2 additional constraints mean the only feasible test cases are those from rows 3, 4 and 6 and here, I can pick row 3 and row 6, both of them do not violate the constraint and they end up satisfying the predicate.



(Refer Slide Time: 25:26)

Recap: RACC on $p = a \wedge (b \vee c)$

- Clause a is the major clause: Only three of the nine sets of test requirements that satisfy will satisfy RACC.
- Row 1 is paired with row 5, row 2 is paired with row 6 and row 3 is paired with row 7.

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
5	F	T	T	F
2	T	T	F	T
6	F	T	F	F
3	T	F	T	T
7	F	F	T	F

- RACC is infeasible for a in the predicate p .

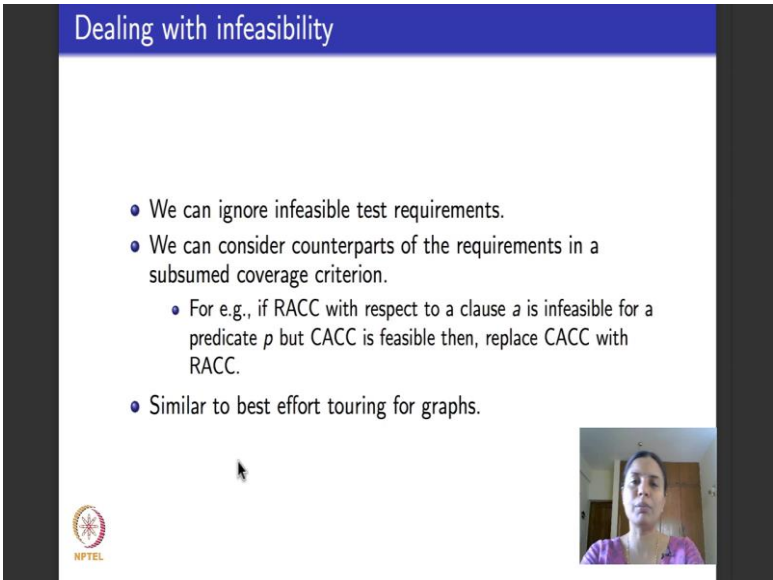



So, CACC is a feasible TR for the predicate p in the presence of those 2 additional constraints. Similarly now let us go back and look at RACC restricted active clause coverage criteria. Here is what we did in the last lecture to be able to derive at test cases for RACC. So, after working out we realize that one of these combinations: row 1 plus row 5, row 2 plus row 6, row 3 plus row 7 would suffice as a good set of test cases for restricted active clause coverage criteria.

But now if you add these 2 extra constraints that were given here which resulted in only rows 3, 4 and 6 being possible candidates for test cases. You see RACC can never be met because it says one should be paired with 5, both are ruled out because one violates both the constraints, 5 violates constraint 2 and it says the next possible combination you can consider is pairing 2 with 6, 6 is fine because it does not violate any constraint, but 2 violates constraint one.

So, that is also ruled out. The next possible combination of test cases for RACC could be pairing of rows 3 and 7. Again if you go back 3 is fine, it does not violate any constraint, but 7 violates constraint one. So, none of the three combinations that we had listed as possible test cases to achieve RACC criteria now for this example, in the presence of constraint are and all 3 of them become unusable. So, we say RACC is infeasible for the predicate for the predicate p when 'a' is the major clause because there are 2 additional constraints in the system. So, like this when you take logic coverage criteria and try to apply it for practical examples because of the presence of the additional constraints in the system some of the coverage criteria requirements could become infeasible.

(Refer Slide Time: 27:20)



The slide is titled "Dealing with infeasibility" in a blue header. It contains a bulleted list of strategies for handling infeasible test requirements. In the bottom right corner, there is a small video inset showing a woman speaking. The NPTEL logo is in the bottom left corner.

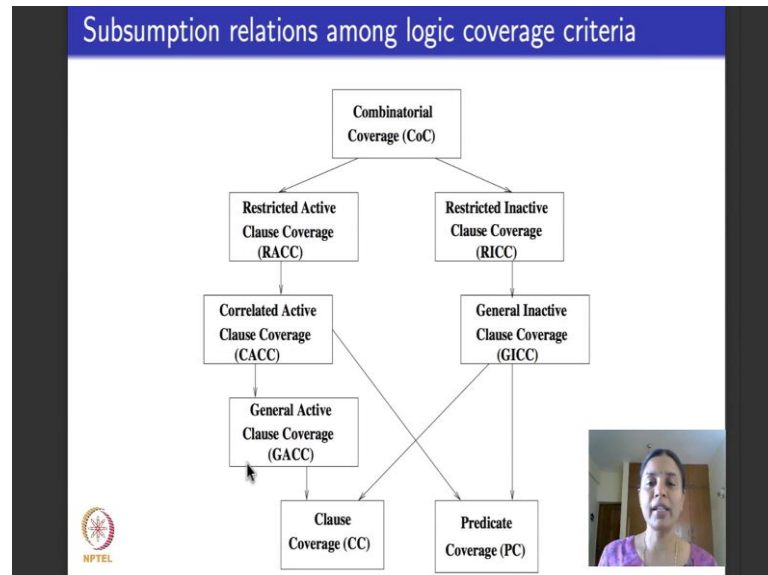
- We can ignore infeasible test requirements.
- We can consider counterparts of the requirements in a subsumed coverage criterion.
 - For e.g., if RACC with respect to a clause a is infeasible for a predicate p but CACC is feasible then, replace CACC with RACC.
- Similar to best effort touring for graphs.

So, hopefully this example would have helped you to understand how infeasibility of these test requirements come into picture. So, how what can we do with infeasible requirements the simplest recommendation that people give as far as testing is concerned is that ignore them and move on ignore the infeasible requirements or we can consider other counterparts of the requirements like for example, if we in this case we realize that RACC is infeasible, but CACC was feasible.

So, you ignore RACC, but consider CACC right. If you remember this is somewhat similar to doing best effort touring graphs. So, whenever we say prime path coverage is infeasible you consider achieving the same thing using side trips and detours. So, you

ignore and you try to replace it with other coverage criteria that would be feasible. That is how you deal with infeasibility in logic coverage.

(Refer Slide Time: 28:16)



So, this is the final slide. What I have done here is I have given you the subsumption relation for coverage criteria. If you remember the definition of what is subsumption, we say one coverage criteria subsumes another coverage criteria if the set of all test cases that satisfy criteria 1 also satisfy criteria 2. So, we realized through an example in the last class that clause coverage and predicate coverage do not subsume each other. There are predicates for which you can achieve predicate coverage, but you may not exercise each clause. There are formulas for which you can achieve clause coverage, but the predicate will be true in both the cases.

So, there is no containment here, they are separate. We know that generalized active clause coverage subsumes clause coverage because it exercises each clause to be true or false and we saw through an example that generalized active clause coverage does not subsume predicate coverage. So, I have not put any arrow here and we also saw that all the ICC criteria, just a little while ago in today's lecture I told you that all the ICC criteria subsume predicate coverage because they test for the predicate to be true and they test the for predicate to be false. And we saw in the last class that CACC subsumes predicate coverage. I know that RACC subsumes CACC, which in turn subsumes generalized ACC this directly follows by definition. Combinatorial coverage is the master of all

coverages because it says you test the predicate for every combination of true false value exponential in number. So, it subsumes all other logical coverage criteria.

So, this is how the various coverage criteria are related amongst each other. So, what we will do in the next module is I will tell you suppose given a predicate, given a set of clauses how to make a clause determine a predicate. Are there algorithms, methodologies to be able to do that. So, that will be the end of week 5 lecture for us.

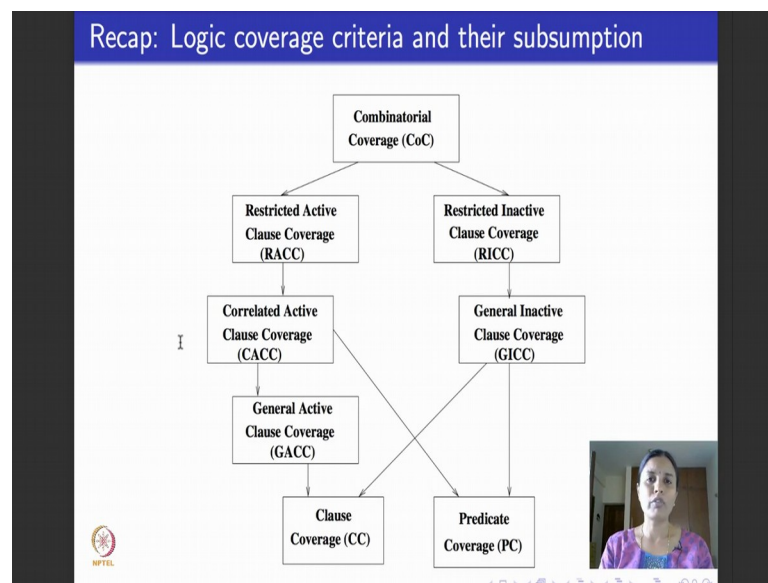
Thank you.

Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 24
Logic Coverage Criteria: Making clauses determine predicate

Hello again. We are in the last lecture of week 5, we are in the middle of doing logic coverage criteria based testing. I defined what the various logic coverage criteria are and showed you the subsumption relations. Today what we are going to see is suppose you have to take a real program and you use this logic coverage criteria to actually correctly write your test requirements and subsequently define test values; how would you go about doing it?

(Refer Slide Time: 00:41)



So, just to recap, what we saw till now, here is all the logic coverage criteria that we saw. We began with predicate and clause coverage, the most elementary form of logic coverage. Predicate coverage says test for the predicate to be true once and false once, clause coverage says test each clause to be true and false once. Then we saw all combinations coverage, right on top in terms of subsumption. It says test for the entire truth table of all the clauses in the predicate. We say that is too expensive because it reads to an exponential number of test cases. So, what can we do in between?

The in-between coverage criteria are broadly classified as active coverage criteria on the left hand side, 3 kinds and inactive coverage criteria on the right hand side, 2 kinds. Both these cases, the premise first choose one clause at a time in the predicate, call it the major clause. In active coverage criteria, major clause determines the predicate, the minor clauses take values as the major clause determines the predicate. In inactive coverage criteria the minor clauses take values such that the major clause does not determine the predicate. So, this is a summary slide that contains all the logic coverage criteria that we saw and how they are all related to each other, what subsumes what and what does not subsume what?

(Refer Slide Time: 02:06)

The slide has a blue header with the title "Test cases for predicate, clause and combinatorial coverage". Below the header, the letter "I" is displayed. A list of four bullet points follows:

- Test cases for satisfying TRs for predicate, clause and combinatorial coverage are easy to define.
- As long as a predicate or a clause is not valid/not a contradiction, we can find test cases to achieve predicate coverage and clause coverage.
- For combinatorial coverage, test cases will correspond to all the values in the truth table.
- The usual problem of reachability (RIPR criteria) will have to be solved for writing test cases.

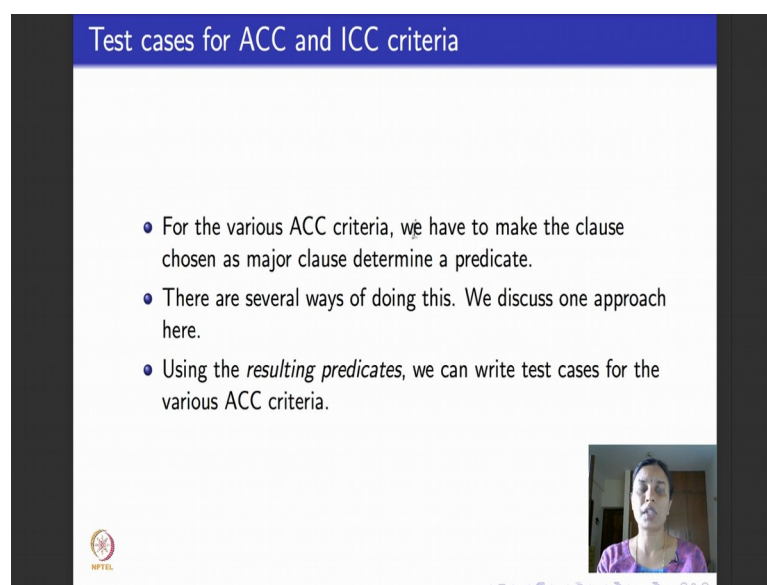
In the bottom right corner of the slide, there is a small video inset showing a woman with dark hair wearing a purple patterned top. The NPTEL logo is visible in the bottom left corner of the slide area.

The focus of today's lecture would be to design test cases. So, we take one at a time and then see how to design test cases. We saw during the logic coverage criteria lecture that we did as a third lecture of week 5, these, how to design test cases for clause and predicate coverage and for all combinations coverage. They are reasonably easy because as long as a predicate is not always true, which means a predicate is not valid or as long as the predicate is not always false, which means the predicate is never valid which means it is a contradiction. So, it is always possible to get test cases that will do coverage criteria for both predicate coverage and clause coverage. You have to identify a set of values that will make the predicate or the clause true once and another set of values that will make the predicate or the clause false once.

For all combinations coverage it is again easy. Write out the entire truth table and then write test cases for every row in the truth table that will make each of the clauses true and false in turn. The only problem that we might have to solve is suppose there is a predicate that comes deep down in a program as a part of an if statement which is way down in a program in the sense that it comes, let us say, after a few hundred lines in the program. It so happens that the variables that you encounter in the if statement, none of those variables deal with input values directly. So, I should be able to give inputs to the program first of all to reach that if statement and exercise my desired coverage criteria for the predicate in that if statement.

So, you remember right in the first week we saw this thing of observability and controllability and then I told you about RIPR criteria. So, we will have to do reachability and then we will have to do infection to be able to achieve our coverage requirement and in turn propagation. So, what we will do in the next week, the first module that we will see we look at logic coverage of source code. Take a piece of program look at all the predicates in the program and see how to apply the coverage criteria that we learnt. At that point we will deal with this last item here of reachability or RIPR criteria in detail which basically means that if a predicate in a program deals with internal variables that are not inputs, how to give input values that will make program reach and achieve the desired coverage criteria on that predicate.

(Refer Slide Time: 04:50)



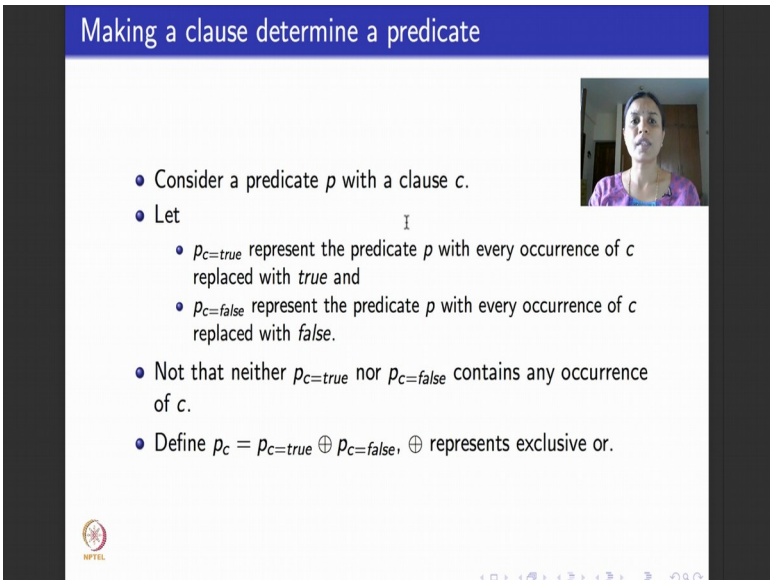
Test cases for ACC and ICC criteria

- For the various ACC criteria, we have to make the clause chosen as major clause determine a predicate.
- There are several ways of doing this. We discuss one approach here.
- Using the *resulting predicates*, we can write test cases for the various ACC criteria.

NPTEL

So, now what about designing test cases for the rest of these coverage criteria? For the 3 active clause coverage criteria and the 2 inactive clause coverage criteria. The first thing to do and remember is that for active clause coverage criteria. We chose each clause will take turn as a major clause and the minor clauses take values such as the major clause determines the predicate. Now, what we will do is how to make a major clause that is chosen in any point in time determine the predicate. Now major clause determining a predicate means the predicate should take true or false values as and when the major clause takes true or false values. So, we will see how to do that by using examples and then based on that how to choose TRs and test cases that satisfy the TRs.

(Refer Slide Time: 05:30)



Making a clause determine a predicate

- Consider a predicate p with a clause c .
- Let
 - $p_{c=true}$ represent the predicate p with every occurrence of c replaced with *true* and
 - $p_{c=false}$ represent the predicate p with every occurrence of c replaced with *false*.
- Not that neither $p_{c=true}$ nor $p_{c=false}$ contains any occurrence of c .
- Define $p_c = p_{c=true} \oplus p_{c=false}$, \oplus represents exclusive or.

So, what I will do is I will walk you through several examples that tell you how to make a major clause determine a predicate. In all these examples we do a generic definition based approach of deciding when a major clause determines a predicate. So, here is what we do to make a major clause determine a predicate. So, you consider predicate p and let us say there is a clause c in the predicate p . I want c to be the major clause which means to be able to do active clause coverage criteria, I want c to be able to determine p and when I do inactive clause coverage criteria, I want c to be able to not determine p . For the purposes of this lecture, I will focus on active clause coverage criteria and tell you how to make a particular chosen clause which is the major clause determine a predicate p . So, I fix a predicate p and I pick up a clause c in that predicate, call it the major clause. The idea is when will c determine p ? For small predicates it will be easy.

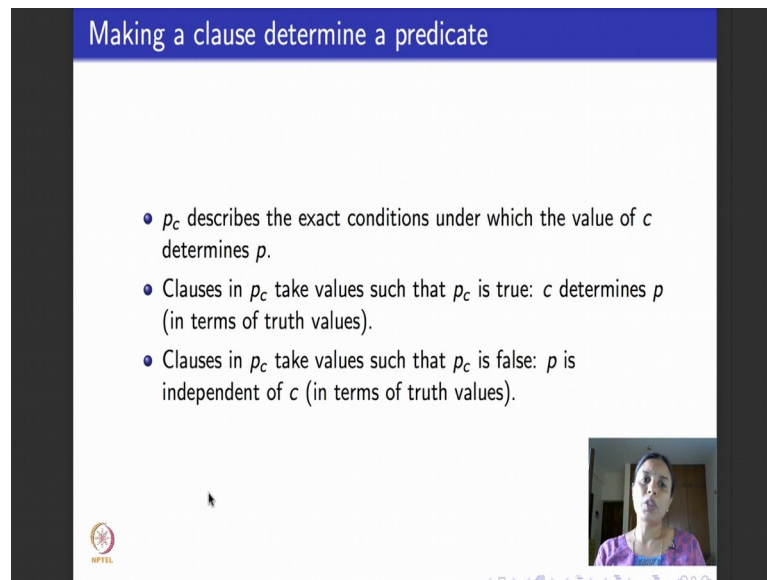
But for large predicates it may not be easy to know when exactly c will determine p . So, we use a default definition based approach which is like a set of steps that you need to do to make c determine p . So, here are the steps. So, what we will do first is we will take the predicate p , take every occurrence of c in p . By the way without loss of generality, I will assume that in the all predicates that we see particular clause occurs exactly once. If it occurs more than once the second occurrence is usually redundant and we remove the second occurrence. So, what I do is I take the predicate p , consider the place where the clause c occurs in the predicate p and replace c with true. So, call that resulting formula or resulting predicate by this notation, p subscript c is equal to true.

Now, I do the second thing that I do is I take the predicate p , consider the clause c replace c with value false wherever c occurs and simplify the logical formula corresponding to the predicate call that p c false. So, is it clear what P_c true and P_c false are? P_c true means take the predicate p , consider an occurrence of c in p replace that occurrence with true and simplify the formula. P_c false says take the predicate p , consider the occurrence of c in the predicate p , replace c with false.

So, once I replace c with true or false to get these 2 predicates P_c true and P_c false, please remember that p has no more occurrences of c . c is completely eliminated from the predicate p . Now I define the following formula P_c . I take this for atomic, this formula P_c true and XOR it with the formula P_c false. Read this notation that looks like a lens of a this thing as XOR. You remember what XOR as a logical operator is? XOR says a XOR b says that it will be true exactly when one of a or b in is true right. So, or says it will be true when one of them is true, it could be the case that both of them will also be true to make an or true. XOR is exclusively or exclusive or means a particular a XOR with b is true if and only if one of a or b is true.

So, I take p , fix a clause c , my major clause, take copy of p replace c with true simplify the predicate keep it on one side that is P_c true. Then I take the same predicate p replace the clause c with false, simplify it, keep it on one side, that is P_c false. Now I XOR these 2 formulas call it P_c .

(Refer Slide Time: 09:37)



Making a clause determine a predicate

- p_c describes the exact conditions under which the value of c determines p .
- Clauses in p_c take values such that p_c is true: c determines p (in terms of truth values).
- Clauses in p_c take values such that p_c is false: p is independent of c (in terms of truth values).

NPTEL

What do we do with P_c ? It turns out that P_c describes the exact conditions under which the value of c determines p . Why is that so? That is so because if you see the clauses that occur in P_c first of all c is not there they take values such that when p is P_c is true c will determine p in terms of its truth values and the clauses in P_c take values such that when P_c is false p .

So, the truth or the falsity is independent of truth or falsity of c . Why does this happen? Instead of doing a proof, I will show you several examples of how to take a predicate and a clause how to determine P_c and understand why P_c will represent the conditions under which the clause c determines P .



(Refer Slide Time: 10:26)

Making a clause determine a predicate: Examples

- Consider $p = a \vee b$.
- Then,

$$\begin{aligned} p_a &= p_{a=\text{true}} \oplus p_{a=\text{false}} & (1) \\ &= (\text{true} \vee b) \oplus (\text{false} \vee b) & (2) \\ &= \text{true} \oplus b & (3) \\ &= \neg b & (4) \end{aligned}$$

- For major clause a to determine p , the only minor clause b must be false.
- Symmetrically, $p_b = \neg a$.



So, here is the first formula. So, the predicate that I want to illustrate by using an example is this predicate. I have taken the predicate p which is $a \vee b$. How many clauses are there in the p , in the predicate p ? There are 2 clauses: one clause is a one clause is b . Now I want to be able to compute p_a which means I fix a to be my major clause and I want a to determine p . So, as per my formula, what should I do? I take this p and first replace every occurrence of a in the predicate p by true and then I replace every occurrence of a in the predicate p by false and XOR them. That is what I have written in the first line here. This is as per our definition. p_a is the same as p with a replaced with true, XOR-ed with p with a replaced with false right. What is p with a replaced with true? p in this example is $a \vee b$. So, you take this predicate $a \vee b$, instead of a write true, that is what I have written in the second line here.

So, this is $a \vee b$ which is the predicate p with a replaced with true and p with a replaced false is false or b with a replaced with false and then my goal is to XOR them. Now what is the semantics of $\text{true} \vee b$? $\text{true} \vee b$ should basically be the same as true right because true is always true. So, b does not really influence. So, $\text{true} \vee b$ simplifies to be true logically and $\text{false} \vee b$ simplifies to be b logically.

So, you can look up the basic rules of inference of proposition and logic and these would be some elementary rules. So, this is what is called the generalization, this is what is

called absorption. So, $\text{true} \vee b$ simplifies to true, $\text{false} \vee b$ simplifies to b . Now again one more rule. I have to XOR true with b which is the same as $\neg b$ right because it says what either exclusively true will be true $\vee b$ will be true right. So, it should be the case that not b is true or in other words b should be false.

So, what I have done here? I have taken the predicate p , I want a to be the major clause in p and I want to compute $p \wedge a$. $p \wedge a$ says, $p \wedge a$ is true whenever a determines p . So, I did use the formula that we had and simply substituted simplified, and got $p \wedge a$ to be $\neg b$ right. Now if we see how do I read this, as I read this as follows: for the major clause a to determine the predicate p the only minor clause b ; b is the only other clause here must be false.

Why is that so? Even without working this out if you independently see this formula here I want a to determine p which means what I want p to be true or false exactly when a is true or false right. Suppose b was true, then, the whole predicate will become true independent of the value of a . So, b has to be false for a to influence when p becomes true and when p becomes false. That is exactly the conclusion that we have derived by using this formula and substituting. We have derived that $p \wedge a$ is the same as $\neg b$ or, in other words, for major clause a to determine the predicate p , $\neg b$ should be true or b should be false. Symmetrically suppose b was a major clause, for b to determine p if I replace the same derivation, in the same derivation a with b , I will get $\neg a$, which means what for b to determine p , a should be false.

(Refer Slide Time: 14:15)



Making a clause determine a predicate: Examples

- Consider $p = a \wedge b$.
- Then,

$$\begin{aligned} p_a &= p_{a=true} \oplus p_{a=false} & (5) \\ &= (true \wedge b) \oplus (false \wedge b) & (6) \\ &= b \oplus false & (7) \\ &= b & (8) \end{aligned}$$

I

- For major clause a to determine p , the only minor clause b must be true.
- Symmetrically, $p_b = a$.



So, we look at a couple of more examples. The second example that we consider will be the predicate p which is now $a \wedge b$ instead of $a \vee b$ which we saw in the last slide. I repeat the exercise, now I want a as my major clause and I want to know when a will determine p . So, I use the formula that we learnt, I say p_a is the same as p with a substituted with true XOR-ed with p with a substituted with false. So, I take p which is $a \wedge b$ here I substitute a to be true. So, I get $true \wedge b$, XOR it with a substituted with false which gives me $false \wedge b$. $true \wedge b$, if I use the rules of logic, maps to b and $false \wedge b$, if I again use the rules of logic, maps to false. Why is this so, because $true \wedge b$ true is more general and is a restrictive operator.

So, it will be true exactly when b is true and false and b will be false all the time because it is anded with false right. So, b XORed with false as per rules of propositional logic is equivalent to b . So, what have we concluded? If a is a major clause, for a to determine p sorry a to determine the predicate p , p_a is the same as b which means for major clause a to determine p the minor clause b must be true. So, if you put it back and intuitively look at it, suppose b was true. Suppose b was true then a will completely determine p no because if a is true, p will be true and a is false, p will be false. That is exactly what we have concluded here and symmetrically if I consider b to be the major clause and repeat this exercise for b I will get p_b which is the conditions under which b determines p to be equivalent to a .

(Refer Slide Time: 16:06)


Making a clause determine a predicate: Examples

- Consider $p = a \equiv b$.
- Then,

$$\begin{aligned} p_a &= p_{a=true} \oplus p_{a=false} & (9) \\ &= (true \equiv b) \oplus (false \equiv b) & (10) \\ &= b \oplus \neg b & (11) \\ &= true & (12) \end{aligned}$$

- For any value of b , a determines p without regard to the value for b .

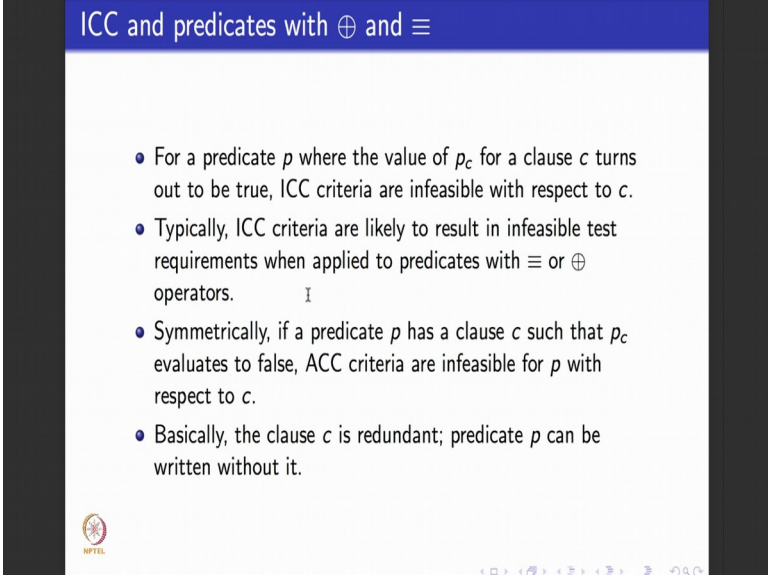
I



So, one more example before we move on. Here I am considering the predicate p to be a is equivalent to b which means $a \equiv b$. I again want a to be the major clause and compute p_a which says when a determines p . So, I use the same formula, take the predicate p , replace the occurrence of a with true XOR it with replace the occurrence of a with false. True equivalent to b turns out to be b , false equivalent to b turns out to be $\neg b$. b XOR with $\neg b$ means what always $b \vee \neg b$ is true which is like a valid formula or a tautology. So, this simplifies to be true.

So, here what has happened? When we try to make a the major clause and make a determine the predicate p , the answer that we get is true. How do you read an answer like true? You read it as no matter what happens, for any value of b if I choose a to be the major clause a can never determine p . So you can come to several conclusions as we saw in these examples. For a predicate like this, I want a to determine p then b must be false. For a predicate which looks like p is equal to a and b if I want a to determine p , then b must be true. For a predicate which looks like a is equivalent to b , I try to make a determine p , but I end up concluding that a can never determine p .

(Refer Slide Time: 17:37)



ICC and predicates with \oplus and \equiv

- For a predicate p where the value of p_c for a clause c turns out to be true, ICC criteria are infeasible with respect to c .
- Typically, ICC criteria are likely to result in infeasible test requirements when applied to predicates with \equiv or \oplus operators.
- Symmetrically, if a predicate p has a clause c such that p_c evaluates to false, ACC criteria are infeasible for p with respect to c .
- Basically, the clause c is redundant; predicate p can be written without it.

NPTEL

So, what, how do I understand this thing of never being able to determine the predicate in terms of testing? In terms of testing, for a predicate p where the value of p_c for a clause turns out to be true, that is what happened here right in this example for the predicate p is equal to a equivalent to b , the value of p_a turns out to be true. In that case it means that a clause cannot determine a predicate right, cannot determine a predicate because the minor clauses cannot take any other values.

So, which means what? Inactive clause coverage criteria is typically infeasible because inactive clause coverage criteria if you remember from the previous lecture, what does it say? It says that minor clauses take values such that major clause does not determine p . But in this case when I end up computing p_a for a to be a major clause and I get the answer true, I conclude that no matter what happens the clause that I have chosen as the major clause cannot determine.

So, I cannot basically do inactive clause coverage criteria. Typically it is known that when I have predicates with XOR or equivalence, ICC criteria turn out to be infeasible because I will end up with these funny results like true and all that. Symmetrically if you think of it a little bit, if I try to compute similar thing p_a and then suppose I end up with false, then I can conclude that active clause coverage criteria will not be feasible. That is what I have written here in the third point. I say if a predicate p has a clause c such that p_c evaluates to false then active coverage criteria will be infeasible for p with respect to c .

It basically means the clause is redundant you might as well remove it from the predicate p.


(Refer Slide Time: 19:32)

Redundant clause in a predicate: Example

- Consider the predicate $p = a \wedge b \vee a \wedge \neg b$. This is just $p = a$.
- Computing p_b , we get

$$\begin{aligned}
 p_b &= p_{b=true} \oplus p_{b=false} & (13) \\
 &= (a \wedge true \vee a \wedge \neg true) \oplus (a \wedge false \vee a \wedge \neg false) & (14) \\
 &= (a \vee false) \oplus (false \vee a) & (15) \\
 &= a \oplus a & (16) \\
 &= false & (17)
 \end{aligned}$$

- It is impossible for b to determine p .



So, I will show an example here. When do I say the clause is redundant? Purposefully in this example I have considered a predicate that looks like this. So, what is the predicate read as? It reads as a and b or with a and not b right. So, now, what is this basically? If you use the rules of logic, propositional logic a little bit and try to simplify it b or not b are neutral, they neutralize each other they will result to be true and a you just have 2 copies of a which are redundant copies of a. So, p the predicate p is basically just a, but for some reason it occurs in complicated form like this. So, if the predicate p is just a, but b is occurring is a sort of a useless form here, I want to be able to conclude that b cannot determine p.

So, how do I do that? I compute p_b and I have to end up with false. That is what this derivation illustrates. What does this derivation illustrate? It says p_b is you do the same formula, replace b with true XOR it with b replaced with false. So, I take the predicate p which is this I replace every occurrence of b with true which is what I have done here and here I replace every occurrence of b with false which is what I have done here and simplify this formula. So, if you simplify this formula $a \wedge true \vee a \wedge \neg true$, you will get a or false and here if you simplify this formula, you will get false or a a or false is the same as a false or a is the same as a a XOR-ed with a is basically false. So, it is

impossible for b to determine p. So, it is as good as saying this b here which occurs in the predicate p is fairly useless, is redundant you might as well remove it.

(Refer Slide Time: 21:26)

Clauses determining a predicate: More examples

- Consider $p = a \wedge (b \vee c)$.
- Then,

$$p_a = p_{a=true} \oplus p_{a=false} \quad (18)$$

$$= (true \wedge (b \vee c)) \oplus (false \wedge (b \vee c)) \quad (19)$$

$$= (b \vee c) \oplus false \quad (20)$$

$$= b \vee c \quad (21)$$

- a determines p when $b \vee c$ is true.
- Three choices make $b \vee c$ true:
 $(b = c = true), (b = true, c = false), (b = false, c = true)$.
- For CACC: Pick one pair when a is true and another when a is false.
- For RACC: Choose the same pair for both values of a.

So, we will go back and do one more example to understand how a clause determines a predicate because it will be useful for you. I will also give you in the assignment for this week, I will give you a set of exercises where you should work out for at least one more formula, how a particular clause determines a predicate.

So, here is another example. The predicate that I have taken this time is p which reads as $a \wedge b \vee c$. Now I want to determine p a which is basically I have chosen a to be the major clause and I want to determine conditions under which a determines p. So, I use the formula p a is p a true XORed with p a false. So, I replace a with true in this predicate p, I get this. I replace a with false in the predicate p, I get this part. I now use the rules of logic to simplify it. True ANDed with $b \vee c$ is the same as $b \vee c$, false ANDed with $b \vee c$ is the same as false, $b \vee c$ XORed with false is the same as $b \vee c$.

So, now what it says is that for a to determine p, $b \vee c$ must be true. That is not too surprising right because if you see p is a ANDed with $b \vee c$ right. So, unless this whole thing is true a cannot influence p. So, $b \vee c$ must be true. $b \vee c$ must be true, you apply the condition for $b \vee c$ as another formula and find out when it will be true. You use the semantics for or. If you use the semantics for \vee , $b \vee c$ will be true in 3 possible cases:

both b and c are true, which is listed here, b is true c is false or b is false c is true. So, there are 3 choices to make $b \vee c$ true. Now suppose I have to do correlated active clause coverage for a then what can I do ? I will pick one pair when a is true and another pair when a is false. So, I can pick a to be true and let us say I pick this, b true, c true. In which case the predicate p will evaluate to true and now a is false I pick some other pair let us say I pick the pair b false, c true.

So, in which case what will happen? This whole thing will be true and a is false. So, p will turn out to be false. So, a will correctly determine p. Now for restricted active clause coverage criteria what do I have to do ? I take value of a to be true and false again, a is the major clause and I choose the same pair because that is what RACC says right. The minor clauses all take the same value.

So, I could take for example, a to be true and b and c to be true, in which case the whole predicate, if I substitute it back will evaluate it to true. Or I could take a to be false and b and c to be true in which case the whole predicate when substituted back will evaluate to false. So, is it clear? Please, how to do p a, p b and p c. So, to be able to be completely satisfied GACC, RACC, CACC and other criteria for this predicate what I have worked out on this slide is only for a. So, you have to repeat a similar exercise for b being the major clause, similar exercise for c being the major clause and then write all conditions for GACC, CACC and RACC. .


(Refer Slide Time: 25:00)


Clauses determining a predicate: More examples


- Consider $p = a \wedge (b \vee c)$.
- Then,

$$\begin{aligned}
 p_b &= \neg p_{b=\text{true}} \oplus p_{b=\text{false}} & (22) \\
 &= (a \wedge (\text{true} \vee c)) \oplus (a \wedge (\text{false} \vee c)) & (23) \\
 &= (a \wedge \text{true}) \oplus (a \wedge c) & (24) \\
 &= a \oplus (a \wedge c) & (25) \\
 &= a \vee \neg c & (26)
 \end{aligned}$$

- Symmetrically, p_c is $a \vee \neg b$.





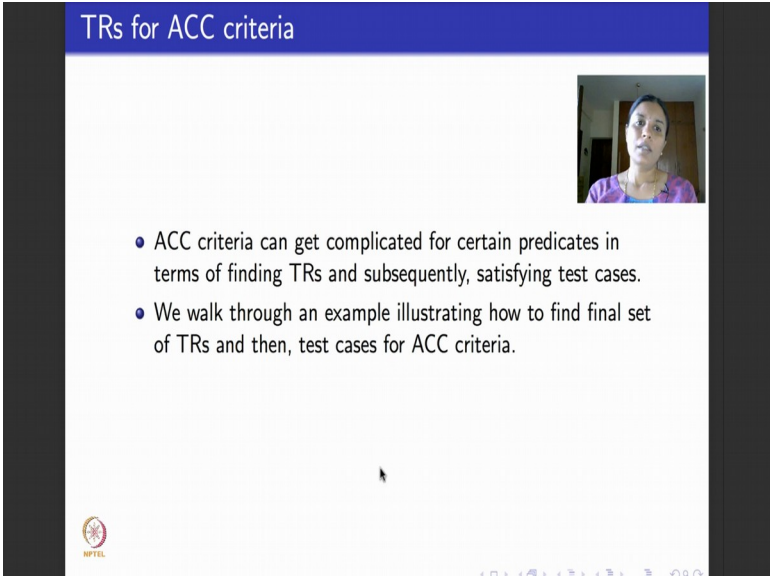


So here for the same predicate, in this slide, I have worked out what happens when b is the major clause. When b is the major clause I am interested in computing $p \wedge b$. So, I take $p \wedge b$ true XOR with $p \wedge b$ false take this predicate p replace b with true I get this part take this same predicate replace b with false I get the part here. Simplify these logical formulae. You might have to remember few rules of propositional logic to be able to do this simplification. So, $a \wedge \text{true}$ is a ; $a \wedge \text{false}$ is same as false, $\text{true} \vee c$ is same as true, $\text{false} \vee c$ is same as c .

So, $a \wedge \text{true}$ is a , $a \wedge \text{false}$ is false, I cannot simplify it further, I keep it like this. Formula $a \wedge b$ XOR with $a \wedge c$ simplified to $a \wedge (b \oplus c)$, the last 2 parts from this line 25 equation 25 to equation 26 may not be very obvious. One simple way to convince yourself that 25 and 26 are the same would be to write the truth table for 25 and write the truth table for 26 and check that the truth tables are the same. That is the easy way to understand why this simplifies to this. Even otherwise if you do not know this simplification it is to leave it like this. $a \wedge (b \oplus c)$ and say this is the condition under which b determines p right because that also be equally handled in terms of writing test cases.

So, similarly because b and c come with this disjunction inside the bracket I can do a similar exercise for P_c and it will be symmetric and P_c will turn out to be $a \vee b$ OR-ed with not b .

(Refer Slide Time: 26:42)



TRs for ACC criteria

- ACC criteria can get complicated for certain predicates in terms of finding TRs and subsequently, satisfying test cases.
- We walk through an example illustrating how to find final set of TRs and then, test cases for ACC criteria.

Now, how to write test requirements for active clause coverage criteria? So, sometimes active clause coverage criteria can get little complicated for certain predicates in terms of finding TRs and subsequently finding satisfying test cases. So, I will walk you through one final example and see how to actually do the end to end TR generation and test case writing for active clause coverage.

So, this time I have taken similar predicate, but slightly different. If you go back, the predicate we had here was $a \wedge b \vee c$.

(Refer Slide Time: 27:20)

TRs for ACC criteria: Example

- Consider the predicate $p = (a \vee b) \wedge c$.
- We first give the truth table for p .

	a	b	c	$(a \vee b) \wedge c$
1	T	T	T	T
2	T	T	F	F
3	T	F	T	T
4	T	F	F	F
5	F	T	T	T
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

- Clauses for p_a , p_b and p_c are given below.

p_a	$\neg b \wedge c$
p_b	$\neg a \wedge c$
p_c	$a \vee b$

Now, what I have taken is $a \vee b \wedge c$. You could do a similar exercise for that predicate also it does not matter, but just to illustrate I have taken another predicate. What was my goal now? My goal is to be able to, for each of these clauses, 3 clauses taking turns to be the major clause to be able to find test requirements for GACC, CACC and RACC. So, there are 3 clauses in this predicate a , b and c . So, what I do is I first write out the truth table of the predicate. It helps to write out the truth table because you can understand it little easily.

So, I have written out the truth table here. Truth table has eight rows, you can work it out for yourself as a small exercise and I have done using this kind of working out that we did for this formula I have calculated p_a , p_b and p_c . I have not shown you the working please try out as a small exercise, but if you use the formula p_a is p_a replaced with true

XOR-ed with a replaced with false and then use the rules of logic to simplify it you should get something like this.



(Refer Slide Time: 28:33)

TRs for GACC criteria: Example

- TR for GACC: Each major clause be true and false, minor clauses be such that major clause determines the predicate.
- TR for GACC: $\{(a = \text{true} \wedge p_a, a = \text{false} \wedge p_a), (b = \text{true} \wedge p_b, b = \text{false} \wedge p_b), (c = \text{true} \wedge p_c, c = \text{false} \wedge p_c)\}$.
- The following table gives true/false values for TR for GACC:

	a	b	c	p
$a = \text{true} \wedge p_a$	T	F	T	T
$a = \text{false} \wedge p_a$	F	F	T	F
$b = \text{true} \wedge p_b$	F	T	T	T
$a = \text{false} \wedge p_b$	F	F	T	F
$c = \text{true} \wedge p_c$	T	F	T	T
$c = \text{false} \wedge p_c$	F	T	F	F

- First and fifth rows are identical, second and fourth rows are identical. Only four tests are needed to satisfy GACC.

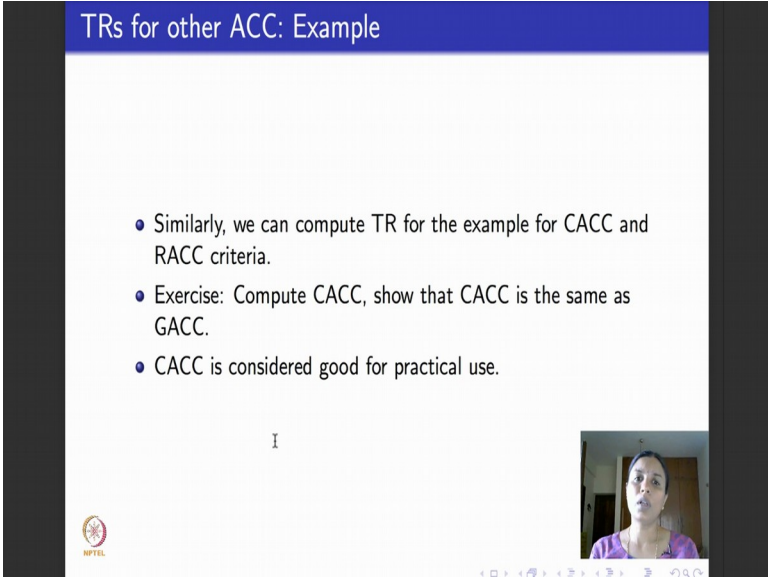
Similarly, for p b, similarly for p c. Now what I am going to do I want to be able to do first do test requirements let us say for generalized active clause coverage criteria. So, let us recollect the definition of GACC. GACC definition says each major clause will be true and false and minor clauses will be such that the major clause determines the predicate. So, now, what will be the TR for GACC ? I am writing what I wrote in English here in terms of true false values. So, the first pair here is when a is the major clause, a is true once AND-ed with a determining p which is p a; a is false AND-ed with a determining p which is again p a. Now the second pair here is b's turn to be the major clause b is the major clause.

So, b is made true once AND-ed with p b which says when b determines p b is made false once, again AND-ed with p b. The third pair here is c's turn to be the major clause: c is true once, false once. For c to determine p, I specify it using p c. So, here is a table that gives you the true false values for the test requirements for GACC. So, a is the major clause first 2 rows I am sorry there is a typo here it should be b is the major clause for the third and fourth row c is the major clause for the fifth and sixth row right. So, a is true once if you see concentrate on the ones in bold in the table a is true once false, once b is true once false once, c is true once false once. The rest of the values take val rest of the

clauses, b and c in this case and a and c in this case and a and b in this case take values such that the predicate becomes true once false once, true once false once, true once false once. So, this is how I do GACC for the predicate p.

Now, if you look at this table you will see that there are repetitions like for example, the first row is the same as the fifth row; a is true, b is false, c is true, a is true, b is false c is true because the assignments are the same, the predicate evaluates to be true. Similarly in the table, the second and the fourth rows are the same. So, I do not have to repeat them. So, I keep only one copy for each of these. So, how many tests totally are needed ? Four test cases are needed one for row 1, one for row 2, one for row 3 and one for row 6 which is not a repeat of any of the earlier rows. So, totally my test requirement for GACC for this example predicate p for each clause a, b and c taking turns to be the major clause can be achieved by the set of four TRs which are basically these rows in the table.

(Refer Slide Time: 31:32)



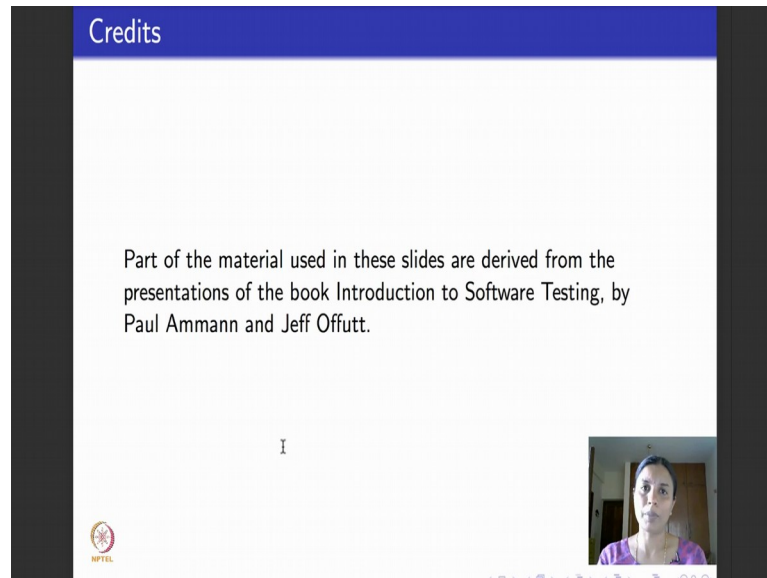
TRs for other ACC: Example

- Similarly, we can compute TR for the example for CACC and RACC criteria.
- Exercise: Compute CACC, show that CACC is the same as GACC.
- CACC is considered good for practical use.

Similarly, you can work out TRs for CACC and for RACC just as a small exercise because it may not be very obvious to do. I urge you all to please work out the test requirements for CACC and for RACC and feel free to get back in touch with me in the forum if you have any doubts about working out. You will realize after working out this particular example correlated active clause coverage criteria is the same as general active clause coverage criteria. Typically for several predicates CACC is considered to be very

good to use it is practically useful and quite useful thing when compared to other predicates.

(Refer Slide Time: 32:15)



So, this will be the last lecture of week 5. What I will do next week? The first 2 modules we will continue with logic coverage criteria, but we will see how it is practically useful. In the next lecture we will take a piece of program and see how to apply logical coverage criteria that we learnt to test that program. And in the lecture after that, we will take a specification as a finite state machine and see how to apply logical coverage criteria that we learnt to be able to test that specification, write test cases for that specification.

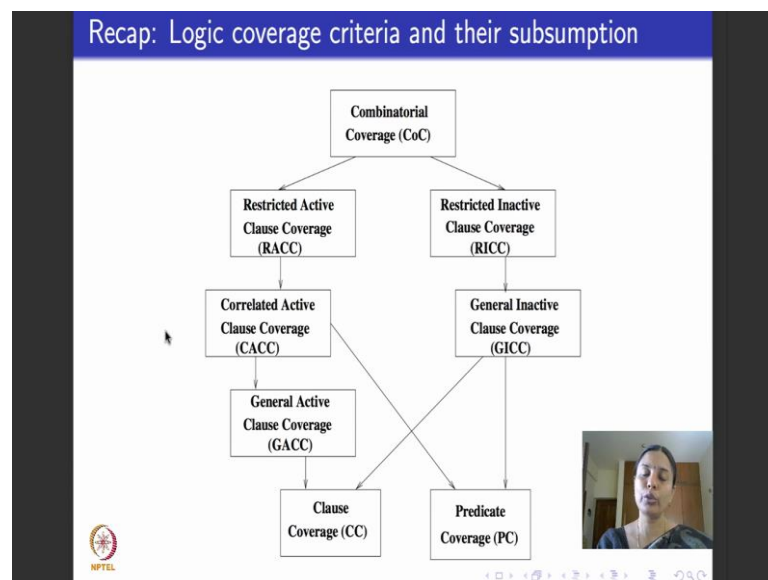
Thank you.

Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 25
Logic Coverage Criteria: Applied to test code

Welcome to week 6, this is the first lecture of week 6. We will continue with logic coverage criteria the whole of this week. Last week we introduced logic coverage criteria, I introduced you to the basics of logic. So, all the coverage criteria, the active clause inactive clause coverage criteria elementary ones, like predicate, clause coverage and then I showed you what was this subsumption relations.

(Refer Slide Time: 00:32)



So, this slide is a recap from last week where we saw all these coverage criteria. To begin with predicate coverage, clause coverage then we saw three categories in active clause coverage, two categories in inactive clause coverage. Then we saw this for the sake of completeness, all combinations coverage which is not going to be useful practically at all. What we will see today is, take source code, see how these coverage criteria that we saw can be used to test for source code.

So, in today's lecture I will show you one example of testing using logic coverage criteria for source code. In the next lecture I will show you another example of testing using logic coverage criteria for source code. I decided to do two examples of source

code unlike graphs because logic coverage criteria is quite predominant and popularly used. So, it needs some amount of practice to be able to understand how to put the coverage criteria that we learnt to practical use.

We will do 2 examples of source code this lecture and next lecture after that I will tell you how to look at logic coverage criteria for things like design constraints exactly like we saw for graph coverage criteria, we will look at pre conditions examples and see how logic coverage criteria applies to them. Finally, we will take specifications as modeled as finite state machines with guards and apply logic coverage criteria to guard so then to see how it works. So, that is the plan for this week.

(Refer Slide Time: 02:06)

Logical predicates from source code

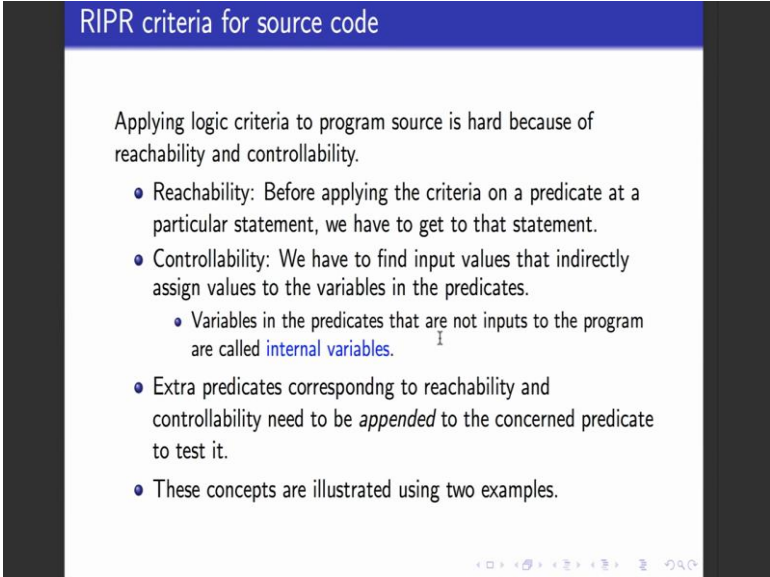
- Predicates are derived from **decision statements**.
 - if-then-else, switch-case, while-do, do-while, for etc.
- In programs, most predicates have less than four clauses.
- When a predicate only has one clause, COC, ACC, ICC, and CC all collapse to predicate coverage (PC).

So, today we begin with source code. When we consider source code where do logical predicates comes from ? They come from every kind of decision statement and the source code which leads to branching in the source code.

For example they could come from if then else statement, it is labeled by predicate every switch case statement is labeled by a predicate statements like while do, do while, do until, repeat until all of them are labeled by predicates, for statements have predicates right. So, programs are started with the decision statements and every decision statement has a predicate in it. So, when I have a predicate corresponding to a decision statement I can apply all the logic coverage criteria that we have learnt to be able to test the program. Typically most programs will have less than four clauses.

In fact, we look at this is the number of clauses in the program and how it matters and then how to apply logical coverage criteria in detail in the third lecture, but typically its consider a good practice or empirical studies in software engineering have shown that most programs have 4 clauses. But typically many programs have predicate where the whole predicate itself is just one clause. It may not may or may not have and ors or other Boolean combinations in which case all the various coverage criteria that we learnt here, all of them basically boil down to only predicate coverage right, which means clause coverage, combinatorial coverage, active clause coverage, inactive clause coverage everything is the same as the predicate coverage because the whole predicate is just one clause.

(Refer Slide Time: 03:40)



The slide is titled "RIPR criteria for source code" in a blue header. The main text explains that applying logic criteria to program source is hard due to reachability and controllability. It lists five bullet points: Reachability (getting to a statement), Controllability (finding input values), a sub-bullet for internal variables, extra predicates for reachability and controllability, and a final point about examples. Navigation icons are at the bottom right.

RIPR criteria for source code

Applying logic criteria to program source is hard because of reachability and controllability.

- Reachability: Before applying the criteria on a predicate at a particular statement, we have to get to that statement.
- Controllability: We have to find input values that indirectly assign values to the variables in the predicates.
 - Variables in the predicates that are not inputs to the program are called *internal variables*.
- Extra predicates corresponding to reachability and controllability need to be *appended* to the concerned predicate to test it.
- These concepts are illustrated using two examples.

Now, to be able to do logical predicate coverage where do I find predicates? As I told you here, you find them all in decision statements. The same more problem comes again in the problem of RIPR criteria or reachability and controllability. So, when I have a predicate that is deep down in my program that lets say labels in if statement, it could be that the variables involved in the predicate are all internal variables. There could be no inputs, there could be no outputs, but how do we give test cases? We give test cases in terms of inputs and then we say what is the expected output. But let us say predicate has no inputs, it has only internal variables then you have to be able to make sure that a test case in terms of inputs can be re written in such a way that the predicate can be reached

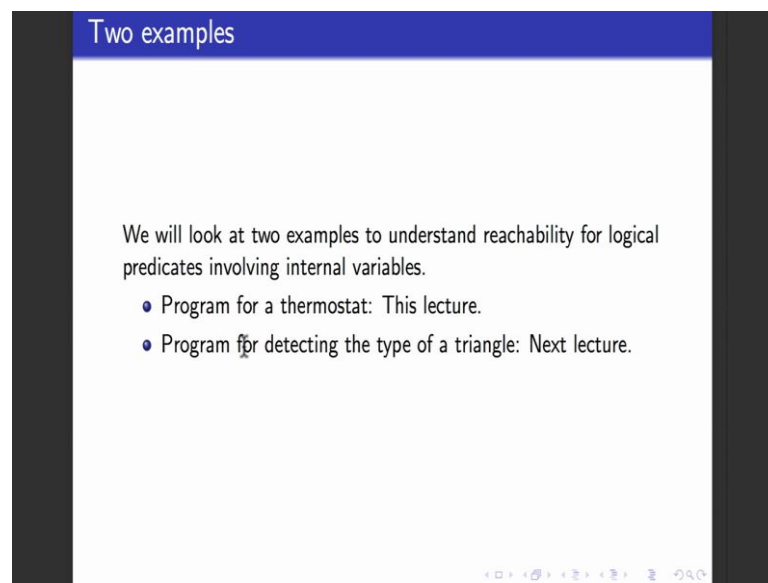
right. You are guaranteed to reach the statement corresponding to the predicate, that is the condition of reachability.

We have to be able to get to that statement meaning, we have to be able to give input values that reach make the program reach the statement by executing all the statements that come prior to that. The next is controllability or what is called infection and propagation. In controllability we have to be able to again give input values in test cases. But in some sense indirectly assign values to the variables that are present in the predicate. Let say there is a particular predicate inside a program that uses all internal variables and I want to test that predicate for predicate coverage. That is make it true once make it false once, but please remember this predicate has all internal variables.

So, to do predicate coverage I should be able to reach this predicate and using input values I should be able to give values to the internal variables such that the predicate is made true once and false once. That is the problem of infection of propagation put together and called as controllability. So, what are internal variables? They are variables in the predicates that do not occur as inputs to the program. So, sometimes to be able to assign values to the internal variables, I need extra predicates where I can add saying that these are the things conditions that have to be satisfied to achieve reachability and controllability and it is the same as adding the conditions as predict as clauses to the original predicate that needs to be tested, because one is to test the predicate I have to be able to reach the predicate and control the predicate. So, I add these as extra conditions.

So, we will see all this through the example of thermostat in today's lecture and then we will see one more example in the next lecture. What is reachability, what is controllability what do we mean by appending extra predicates to the original predicate that corresponds to reachability and controllability, we will see it through we using examples.

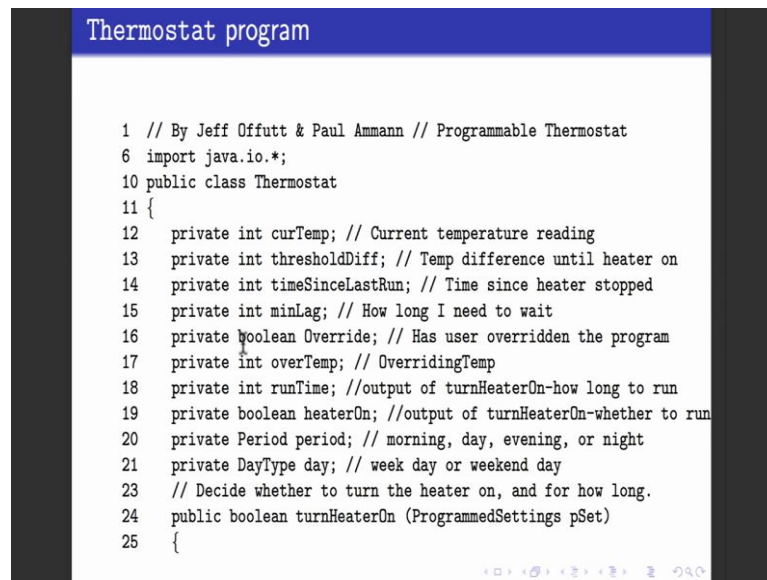
(Refer Slide Time: 06:25)



So, as I told you logic coverage criteria is probably one of the most important testing criteria, very powerful used across board in all kinds of software so it is important to get that correct. So, I will do source code for logic coverage criteria and illustrate the problems of reachability and controllability through two examples. The first example we will be doing is a program for a thermostat.

What is the thermostat? A thermostat is a device that controls the heating of a particular entity. So, there is a small piece of java program that is written for a thermostat. I will show you the program we look at the predicates in the program and see how to apply logic coverage criteria. In the next lecture for this week I will take you through another program which is an elementary high school program which basically takes the inputs as three sides of a triangle and tells you what that what is the type of the triangle? Is this an equilateral triangle isosceles triangle and so, on is a very good old program that not only in this book in this lecture you will find it in several lectures in testing. So, it is a nice thing, a nostalgic thing to be able to go through that program once again. So, what we will do today is we will begin with the thermostat program.

(Refer Slide Time: 07:31)

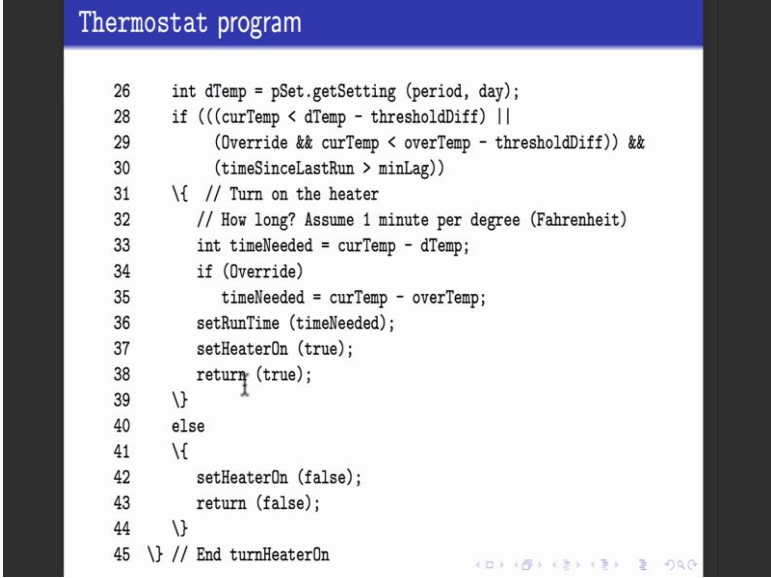


So, here is the code for the thermostat program that is taken from the textbook that we are following the book on Introduction to Software Testing. I have taken the code as it is the authors of the code or the authors of the book as its acknowledged here. So, there is a class called thermostat. What do these lines twelve to twenty one give you they give you all the variables that this program handles. So, there is an integer variable called current temp here right hand side the comment tells you what it represents, it represents the reading of the current temperature it could be the temperature of a particular room right. And then next variable is called threshold difference abbreviated as threshold diff, it tells you what is a temperature difference until the heater gets switched on and then the third integer variables says time since last run, which means what was the time since the heater stopped last time.

So, what is the thermostat trying to do? Maybe its controlling a room in a building in a very cold region it is trying to keep the room heated in a particular ambient temperature. So, this program tells you when to switch it on, when to switch it off and what will be the basic functionality of a thermostat. So, the fourth variable is a Boolean variable called override; override means the user switches off the automatic control in the thermostat and he is trying to configure the temperature setting himself. Then there is one more integer variable called overriding temperature which means when thermostat is in the over written mode what is the temperature that it is been configured for. And there is another integer variable called runtime which is, when to run the heater on, when to turn

the heater on and how long to run it and then there is something called period which tells you whether its morning day evening or night. Maybe it implicitly influences a temperature in which the thermostat will be running and then there is another type enumerated data type called day, which is of the type weekday or weekend day. So, here is the main code corresponding to the thermostat. So, the method that is called is called turn heater on which takes program settings input as ptest to the program. So, there is a pre programmed setting for the thermostat that is turned, that is used to be able to turn the heater on.

(Refer Slide Time: 09:43)



```

26  int dTemp = pSet.getSetting (period, day);
28  if (((curTemp < dTemp - thresholdDiff) ||
29      (Override && curTemp < overTemp - thresholdDiff)) &&
30      (timeSinceLastRun > minLag))
31  \{ // Turn on the heater
32      // How long? Assume 1 minute per degree (Fahrenheit)
33      int timeNeeded = curTemp - dTemp;
34      if (Override)
35          timeNeeded = curTemp - overTemp;
36      setRunTime (timeNeeded);
37      setHeaterOn (true);
38      return (true);
39  \}
40  else
41  \{
42      setHeaterOn (false);
43      return (false);
44  \}
45  \} // End turnHeaterOn

```

So, here is the code corresponding to the thermostat. Please remember I had in all these slides when I show a piece of code I showed across several slides. So, that it becomes visible, but you should, when you read it read it as a piece of code that is meant for continuous execution. As I told you when we did graphs it might be the case that the code as it is may not be taken in readily executable, it could be a fragment of a code.

But we will focus on the things that we need in the code. So, I may not give you the complete code and the code will run through several slides. So, here is the second slide where the code continues, this is the code corresponding to this method that begins in line in twenty four. So, what is this say? This method has another internal integer variable called dTemp which basically tells you some, it ask you to input the steeing, which is the period which is day through this right pSet, and then it has this code. So, it

says there is an; if statement which has several conditions here. So, it says if current temperature is less than d temperature minus threshold difference or read this thing as or override is true and current temperature is less than override temperature minus threshold difference and time since last run is greater than minimum lag.

So, what it says is basically current temperature is, the room is become cold right that is what the first one says. Second one says that the override is on Boolean variable overhead is on which means users trying to make over and run the thermostat and the temperature in the override mode is still minus the threshold, current temperature is still less than that. So, the room is again cold even in the override mode and heat has been off for some time. So, what do I have to do? I have to turn on the heater right, when I say I have to turn on the heater, we also have to specify for how long to turn on and assume that the heater heats at the rate of per degree of heat, per degree Fahrenheit of heating, it takes one minutes to heat, right. So, using this gradation in the temperature of increase in the temperature of heater is specify how long to keep the heater on? If you keep it on for too long maybe the room will become too hot. So, you need to be able to set it. So, that the room temperatures just optimal and comfortable.

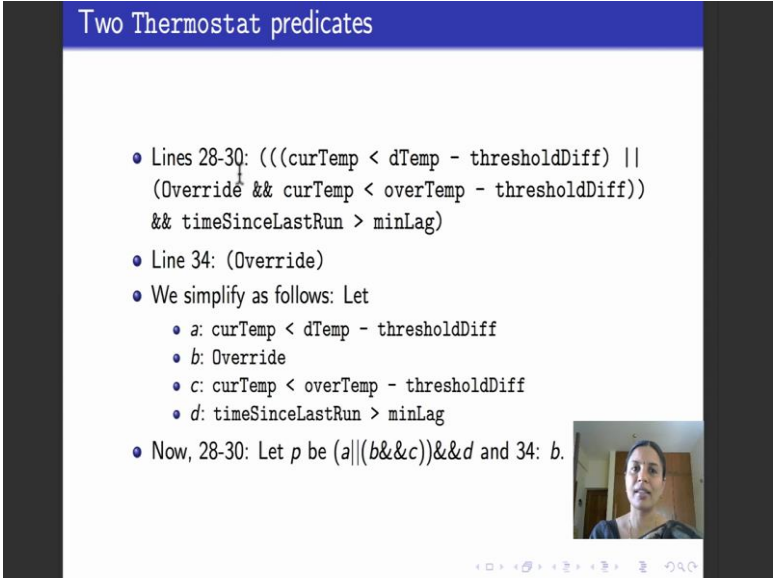
So, this is the code that does that. So, it first computes the time needed to heat it the turn the heater on which is the difference between the temperature in the room and the desired temperature dTemp. And then it says if override is on hen the diff time needed to turn the heater on is the difference between the current temperature. And the overridden temperature and then it says set run time to be time needed, set heater on to be true and then you return true. This is the method that returns a Boolean variable right. Otherwise you said set heater on is false then you return false.

So, is it clear please, what the simple code does? Tts the code corresponding to a thermostat, tries to keep the temperature in a room in a building into optimal setting it uses all these variables, which basically tell you what is the current temperature for how long is the heater been on or off. And whether the heater is been overridden, the override mode being taken by user in which case what do I do. And this is and what sort of it day, it is morning day evening night or is it a week day, weekend maybe weekdays where all the people in the house occupants in the house are away. So, you do not have to turn the heater on maybe in weekends you have to keep it on for longer all these things matter. Then it uses one Boolean method called turn heater on which basically decides whether

to turn the heater on or not. So, that Boolean method has an internal variable dTemp and then it has if temperature, decides how long to turn on the heater on and does this a setting and turns sets the heater on variable to be true. Otherwise it keeps the heater off and returns false.

So, now our goal is to be able to do this program, test this program by applying the logic coverage criteria that we have learnt. So, the first thing to look for when we apply logic coverage criteria is; what are the predicates in the statement the program and where are they? So, you go back and look at the code in this program. The first part of the program is just declarations, no predicates here, obvious. Second part of the program has this long statement here, this one if statement which is this between lines 28 twenty nine 30 and then there is one more if statement here, at line 34 if override.

(Refer Slide Time: 14:12)



Two Thermostat predicates

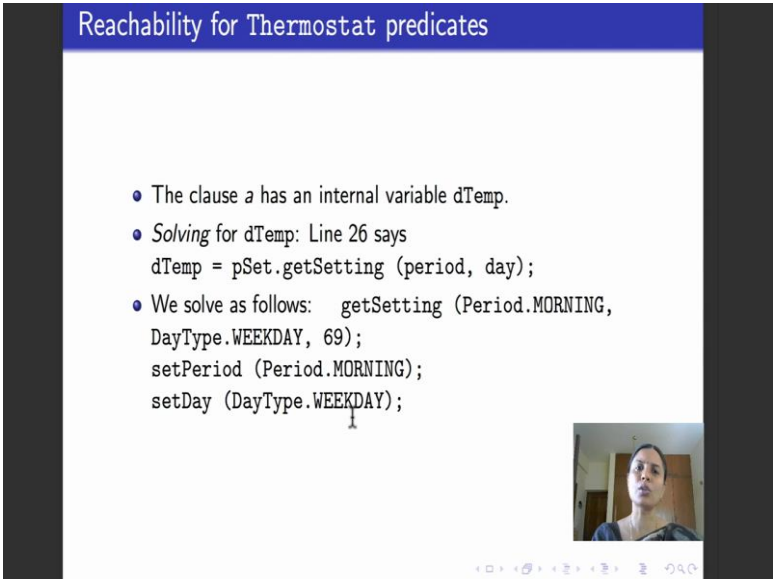
- Lines 28-30: `((curTemp < dTemp - thresholdDiff) || (Override && curTemp < overTemp - thresholdDiff)) && timeSinceLastRun > minLag)`
- Line 34: `(Override)`
- We simplify as follows: Let
 - a: `curTemp < dTemp - thresholdDiff`
 - b: `Override`
 - c: `curTemp < overTemp - thresholdDiff`
 - d: `timeSinceLastRun > minLag`
- Now, 28-30: Let p be $(a || (b \& \& c)) \& \& d$ and 34: b .

So, what I have done in the next slide is that I tell told you what the 2 thermostat predicates are. So, there is a predicates in line 28 to 30. I have just copied the predicate that occurs as label of the if statement here and there is one more predicate called override at line 34. So, this is a pretty long thing right and may be difficult to read. So, let us simplify it we call a as this: current temperatures less than d temperature minus threshold difference. We call b as override, we call c as this clause current temperature is less than override temperature minus threshold difference and we called d as time since last run greater than minLag. So, there are 4 clauses: a which is the first one, b override,

c which is the third one and d which is the fourth one. Now the predicates looks readable in the notation that we have used to understanding it.

What is a predicate? Basically it is a or b and c and d right where a is this first part b is this override c is this third part and d is the fourth part, And then next predicate override is just one clause predicate b. So, this predicate first predicate p which occurs at lines 28 to 30 is interesting for us to test because it has got 4 classes a, b c and d and it is called combination of ors and ands.

(Refer Slide Time: 15:25)



Reachability for Thermostat predicates

- The clause *a* has an internal variable *dTemp*.
- Solving for *dTemp*: Line 26 says
`dTemp = pSet.getSetting (period, day);`
- We solve as follows: `getSetting (Period.MORNING,
DayType.WEEKDAY, 69);
setPeriod (Period.MORNING);
setDay (DayType.WEEKDAY);`

So, let us go ahead and test it for that predicate right. So, now, if you look at this predicate, this predicate has one internal variable called *dTemp*. We will go back to the code for a minute. Remember here *dTemp* is a internal variable that was local to the method turn heater on. The rest of the variables in these predicates are all inputs and outputs. So, as I told you, we have to be able to do reachability first which means we have to be able to solve and make sure that a predicate occurring in these lines is actually reached. Which means what ? In this case because it is a smaller program, it is easy to do solving here means just getting appropriate values for *dTemp* give a value for period and day for *dTemp* and let it assign, let it get assigned to that value and then you will be able to reach this predicate. There are no other conditions. Clause *a* as I told you has an internal variable *dTemp*.

So, I have to be able to solve for dTemp. So, I just give some values. So, this is the line that it is given in the program and look at it. So, it says this pSet is get setting period. So, I give it some value as a period is morning, day as a weekday. So, that is it I have assigned value to dTemp.

(Refer Slide Time: 16:39)

Predicate coverage: p being true

- a: $\text{curTemp} < \text{dTemp} - \text{thresholdDiff}$: true
- b: Override : true
- c: $\text{curTemp} < \text{overTemp} - \text{thresholdDiff}$: true
- d: $\text{timeSinceLastRun} > (\text{minLag})$: true

Now, what I do I take back and these are the 4 clauses a, b, c and d. Please look at the title of the slide it says I am testing for predicate coverage of this clause for the value of the predicate p being true. So, what sort of a predicate is that? That is a predicate that looks like this; it has got 2 ands and an or so, one simple way of making the whole predicate true is to make all the 4 clauses true which means I make a, b, c and d all of them true. That is what I have written here. I say a which is current temperature less than desired temperature minus threshold difference is true b is true, c is true and d is true. So, to make all these things true what should I give I should give values for current temperature such that current temperature values indeed less than desired temperature minus threshold difference.

Override is a Boolean variable. I make it true directly. Similarly to make c true, I should be able to make current temperature true and make the overridden temperature minus threshold difference to be a value greater than the current temperature and so on.

(Refer Slide Time: 17:42)

Predicate coverage: p being true: Test cases

- `thermo = new Thermostat(); // Needed object`
- `settings = new ProgrammedSettings(); // Needed object`
- `settings.setSetting (Period.MORNING, DayType.WEEKDAY, 69); // dTemp`
- `thermo.setPeriod (Period.MORNING); // dTemp`
- `thermo.setDay (DayType.WEEKDAY); // dTemp`
- `thermo.setCurrentTemp (63); // clause a`
- `thermo.setThresholdDiff (5); // clause a`
- `thermo.setOverride (true); // clause b`
- `thermo.setOverTemp (70); // clause c`
- `thermo.setMinLag (10); // clause d`
- `thermo.setTimeSinceLastRun (12); // clause d`
- `assertTrue (thermo.turnHeaterOn (settings)); // Run test`

So, here are a set of values that I gave for the predicate being true. So, I need this object thermostat, I need the object program settings and then I pass values to dTemp by saying period is morning day type is weekday and then I set let say current temperature to be 63, threshold difference to be 5, set some value. Is it clear that current temperature my this a will become true right. Because current temperature will be less than d temperature minus threshold difference why because this is 69 no, and then override has to be true. Now I am working on clause c. I have set overridden temperature to be 70. So, that lag n makes c to be true because current temperature will be less than overridden temperature minus threshold difference, 65 is less than, 63 is less than 65 and then to make clause d true, I set minLag and time since last run such that the difference is, time since last run is greater than minLag.

So, time since last run is twelve minLag is 10. So, it is greater than 10. So, fine, right? So this slide, what is it contain? It contains the final set of test cases that will test which predicate, this predicate p occurring at lines 28 to 33 for predicate coverage being true. So, similarly for predicate coverage being false, I have to be able to give test cases that will make a, b, c and d true or false appropriately. So, for the true for this predicate to be false, what could I do? I should make d false because it is and. Once I make d false, a, b and could e anything because anything anded with false becomes false or I could make this whole thing false along the d being true or false. So, like added here I decide what I want to do in terms of making each of the clauses true or false to make the predicate b


false and then I give values such that they are met. I have not given them to you in my slides, but I hope it is clear how to do it.

(Refer Slide Time: 19:47)

Towards CACC for p

- We illustrate the steps for CACC coverage for a being the major clause for $p = (a || (b \& c)) \& d$.
- Determining p_a :

$$\begin{aligned}
 p_a &= p_{a=true} \oplus p_{a=false} \\
 &= (true || (b \& c)) \& d \oplus (false || (b \& c)) \& d \\
 &= (true \& d) \oplus ((b \& c) \& d) \\
 &= d \oplus ((b \& c) \& d) \\
 &= !(b \& c) \& d \\
 &= (!b || !c) \& d
 \end{aligned}$$
- Similarly, we can determine p_b , p_c and p_d .



Now, for the same predicate in lines 28 to 33 I want to be able to let say I tempt correlated active clause coverage which was another coverage criteria that we saw. You could attempt any other coverage criteria, you could do clause coverage, you could do generalized active clause coverage you could do inactive clause coverage, but I have put the CACC is an example to tell you how correlated active clause coverage is done for p . So, if you remember the lecture from last week, to be able to do correlated active clause coverage, we have to take each clause to be a major clause and first make the major clause determine the predicate. Once the major clause determines the predicate, I write one set of test cases that make the predicate true when the major clause is true and one set of test cases that make the predicate false when the major clause is false or true.

So, this predicate has 4 clauses a , b , c and d . Each of these 4 clauses can take turns to be the major clause and each of them can determine p . So, I will worked out for when a will determine p . If you remember from the last weeks lecture when a determines p will be called p_a . So, p_a was obtainable by using this formula. So, you make a true and xor it with making a false in the same predicate. So, I have reworked this solution for p_a . So, p_a true is what? Substitute a to be true in this predicate. So, you will get true or-ed with b and c and d xor with false or-ed with b and c and d . Now you simplify this, true or-ed

with b and c is nothing but true. So, that b true and d false or with b and c is nothing, but b and c. So, that be b and c and d, I go on simplifying it further then I get this final value.

As I told you in case you are not clear about how to go from these fourth line to fifth line and fifth line to sixth line, why are they equivalent ? You could stop at fourth line or fifth line, they are equivalent. So, in terms of obtaining p_a assuming that you would stopped at fourth line or fifth line that is also good enough. You need not know how to simplify this to be able to get the last line. Even if it is stopped here it is good enough. So, same way we can determine p_b , p_c and p_d . I have not worked it for you, but in the same way you could do that. In fact, you must try and do it as a little exercise. So, now, what I have done is here is how the test requirement for CACC will look like.

(Refer Slide Time: 22:04)

CACC for p

TR for CACC for p for each clause to be major clause:

	a	b	c	d
p_a :	T	t	f	t
	F	t	f	t
p_b :	f	T	t	t
	f	F	t	t
p_c :	f	t	T	t
	f	t	F	t
p_d :	t	t	t	T
	t	t	t	F

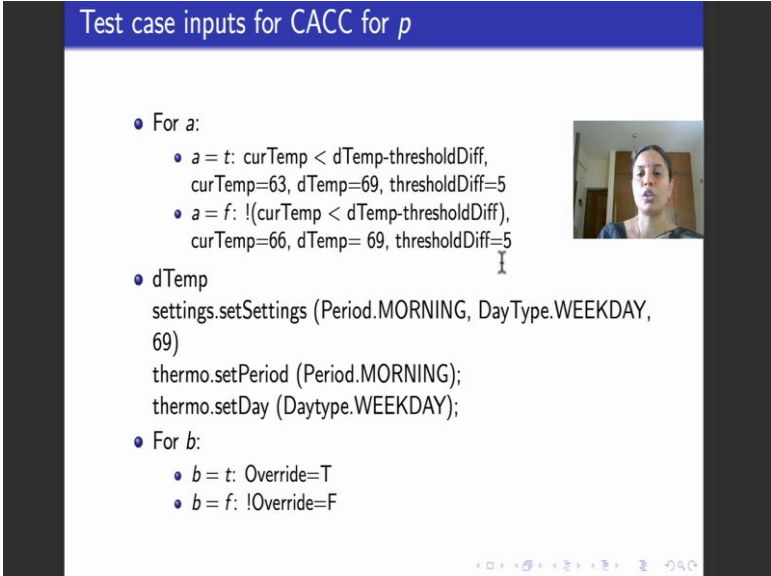
Rows 3 and 5 are duplicates, six tests are needed to achieve CACC TR for p .

So, how do you read this? There are 4 clauses a , b , c and d and this p_a is when a is the major clause, p_b is when b is the major clause, p_c is when c is the major clause, p_d is when d is the major clause. When a is the major clause this capital T and capital f when the column corresponding to a indicate that a is made true once, a is made false once. The rest of the values for b , c and d indicated by small t and small f for true and false tell you the values that the clauses b , c and d take for a to determine p . Similarly for p_b when b determines p , b is made true once false once. The other clauses a , c and d which are minor clauses when b is the major clause take values such that predicate becomes true once and false once. Similarly for c , when c is the major clause the true false values

corresponding to *c* are written in capital letters the rest of the clauses take values such that *c* determines *p* and finally for *d*, right.

So, now if this is how I write and fill up the test cases. Now I examine the table once again to see how many repetitions are there in the table. So, if you see row number three false true, *a* is false, *b* is true *c* is true, *d* is true is the same as row number 5, *a* is false here, *b* is true, *c* is true *d* is true. Please do not worry about capital *T* true being capital *T* here in for *b* and for capital *T* here for *c*. It just still means true. So, these are basically duplicates. So, I do not have to repeat. So, if we ignore the duplicates I basically need 6 test cases to be able to satisfy correlated active clause coverage criteria for *p*. What are the 6 test cases ? The rows that do not come as duplicates, these, all other rows. So, what I do? I take one row at a time and like I did for predicate coverage I give values to all the variables that makes each clause true or false in turn right. So, for *a* to become true, here is the set of values.

(Refer Slide Time: 24:16)



Test case inputs for CACC for *p*


- For *a*:
 - *a* = *t*: *curTemp* < *dTemp*-*thresholdDiff*,
curTemp=63, *dTemp*=69, *thresholdDiff*=5
 - *a* = *f*: *!(curTemp* < *dTemp*-*thresholdDiff*),
curTemp=66, *dTemp*= 69, *thresholdDiff*=5
- *dTemp*
`settings.setSettings (Period.MORNING, DayType.WEEKDAY, 69)`
`thermo.setPeriod (Period.MORNING);`
`thermo.setDay (Daytype.WEEKDAY);`
- For *b*:
 - *b* = *t*: *Override*=*T*
 - *b* = *f*: *!Override*=*F*

So, *a* is this for *a* to become true current temperature could be 63, desired temperature could be 69, threshold difference could be 5. For *a* to become false, current temperature could be 66, desired temperature could be 69 and threshold difference will be 5. In this case current temperature will not be less than desired temperature minus threshold difference. And *dTemp* we saw, you have to pass that value by setting values for morn for a period and for dtype. *b* is very simple because it is just is Boolean variable override.

(Refer Slide Time: 24:53)

Test case inputs for CACC for p , contd.

- For c :
 - $c = t$: $\text{curTemp} < \text{overTemp} - \text{thresholdDiff}$,
 $\text{curTemp}=63$, $\text{dTemp}=72$, $\text{thresholdDiff}=5$
 - $c = f$: $\neg(\text{curTemp} < \text{overTemp} - \text{thresholdDiff})$,
 $\text{curTemp}=66$, $\text{dTemp}=67$, $\text{thresholdDiff}=5$
- For d :
 - $d = t$: $\text{timeSinceLastRun} > \text{minLag}$
 $\text{timeSinceLastRun}=12$, $\text{minLag}=10$
 - $d = f$: $\neg(\text{timeSinceLastRun} > \text{minLag})$
 $\text{timeSinceLastRun}=8$, $\text{minLag}=10$




Similarly, for c , c is this clause I give values for current temperature desired temperature and threshold difference to make c true once false once. Similarly for d . So, once I have this I go back to this table. For every row in the table a true, b true, c false, d true. here is the test case.

(Refer Slide Time: 25:12)

Test cases for CACC, #1

$\text{dTemp} = 69$ (period = MORNING, daytype = WEEKDAY)

- 1. T t f t
`thermo.setCurrentTemp (63);`
`thermo.setThresholdDiff (5);`
`thermo.setOverride (true);`
`thermo.setOverTemp (67); // c is false`
`thermo.setMinLag (10);`
`thermo.setTimeSinceLastRun (12);`
- 2. F t f t
`thermo.setCurrentTemp (66); // a is false`
`thermo.setThresholdDiff (5);`
`thermo.setOverride (true);`
`thermo.setOverTemp (67); // c is false`
`thermo.setMinLag (10);`
`thermo.setTimeSinceLastRun (12);`




Read this as a true, b true, c false, d true in order: a, b, c, d. Here is the test case, this is the complete test case for the first row in this test requirement for CACC.

(Refer Slide Time: 25:30)

Test cases for CACC, #1

dTemp = 69 (period = MORNING, daytype = WEEKDAY)

- 3. f T t t
thermo.setCurrentTemp (66); // a is false
thermo.setThresholdDiff (5);
thermo.setOverride (true);
thermo.setOverTemp (72); // to make c true
thermo.setMinLag (10);
thermo.setTimeSinceLastRun (12);
- 4. F f T t thermo.setCurrentTemp (66); // a is false
thermo.setThresholdDiff (5);
thermo.setOverride (false); // b is false
thermo.setOverTemp (72);
thermo.setMinLag (10);
thermo.setTimeSinceLastRun (12);




Similarly, I give 6 set of test cases right, this is a second one, this is the third one, this is the fourth one where a is false, b is false, c is true, d is true.

(Refer Slide Time: 25:38)

Test cases for CACC, #1

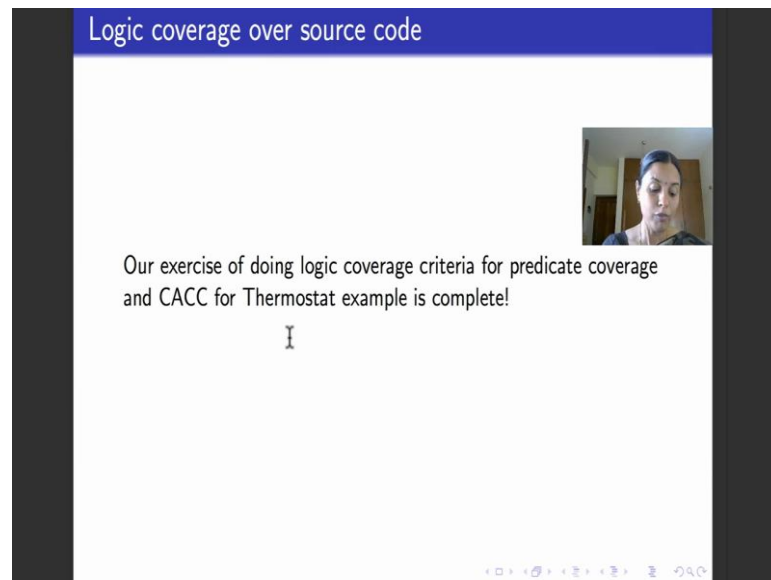
dTemp = 69 (period = MORNING, daytype = WEEKDAY)

- 5. t t t t T
thermo.setCurrentTemp (63);
thermo.setThresholdDiff (5);
thermo.setOverride (true);
thermo.setOverTemp (72);
thermo.setMinLag (10);
thermo.setTimeSinceLastRun (12);
- 6. t t t F
thermo.setCurrentTemp (63);
thermo.setThresholdDiff (5);
thermo.setOverride (true);
thermo.setOverTemp (72);
thermo.setMinLag (10);
thermo.setTimeSinceLastRun (8); // d is false



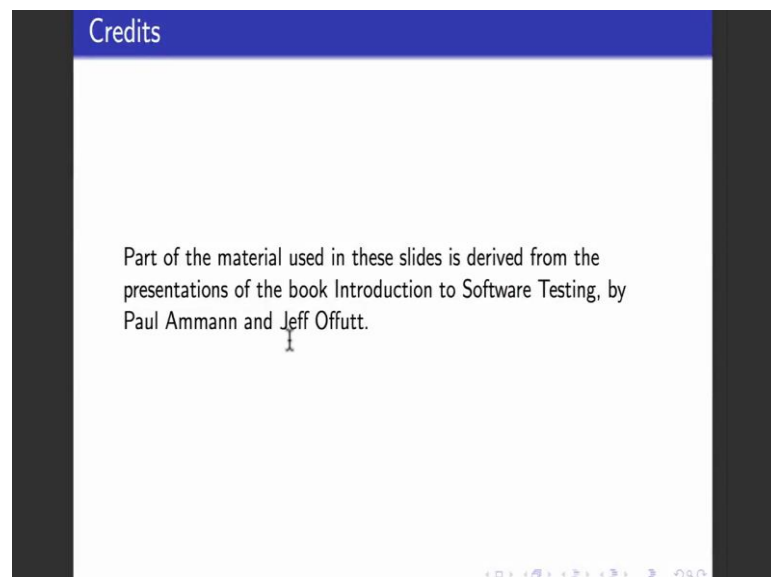
Similarly, this is the fifth one. All 4 are true, this is the sixth one, a, b and c are true and d is false. So, these are the final test case values that will help us to test correlated active clause coverage criteria for the predicate p in lines 28 and 30. So, similarly you could do clause coverage for that predicate, generalized active coverage criteria for the predicate, RACC for the predicate, all other conditions that you want.

(Refer Slide Time: 26:06)



Hopefully this exercise of this example would have helped you to understand how to apply logic coverage criteria for source code and how to write test cases end to end. What we will do in the next lecture is I will take you through another examples slightly longer and more complicated than this.

(Refer Slide Time: 26:32)



So, that you understand reachability and controllability well and tell you how to apply logical coverage criteria for that example.

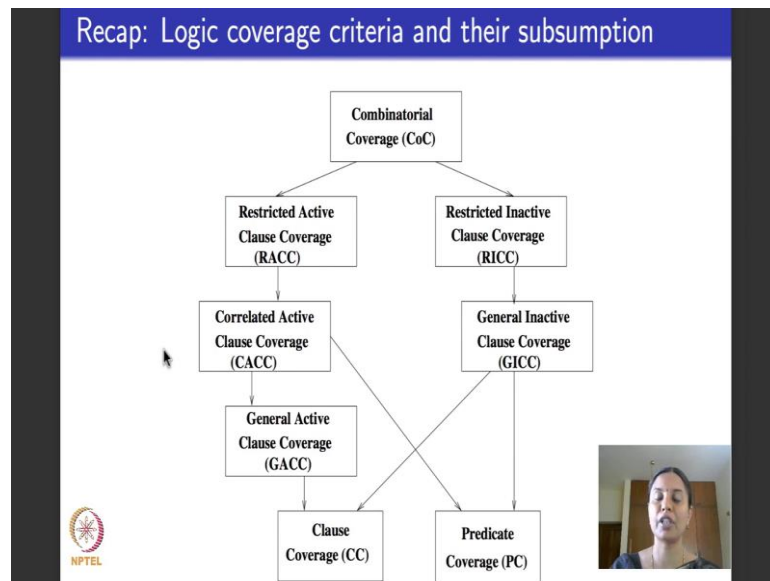
Thank you.

Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 26
Logic Coverage Criteria: Applied to test code

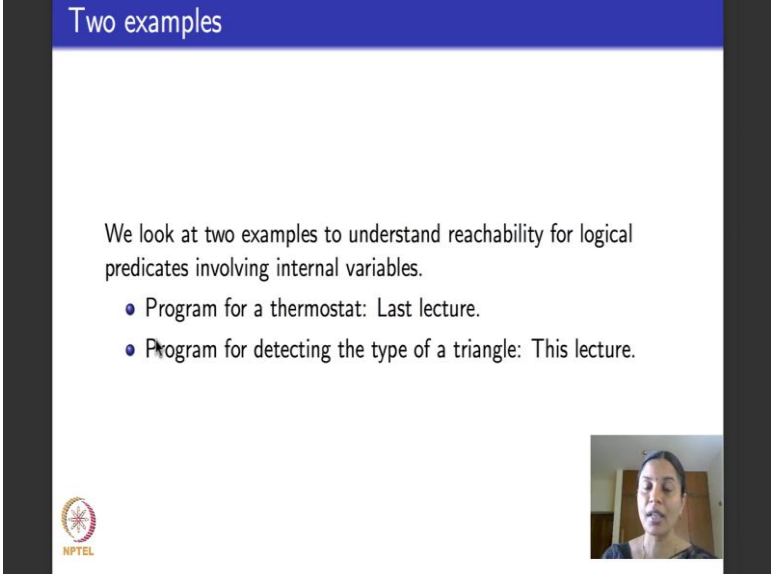
Hello again, welcome to the second lecture of week 6.

(Refer Slide Time: 00:26)



So, what are we going to do today? In this lecture, we are going to look at logic coverage criteria again to test code. So, if you remember last time I told you that logic coverage criteria when it comes to testing code is very important.

(Refer Slide Time: 00:29)



Two examples

We look at two examples to understand reachability for logical predicates involving internal variables.

- Program for a thermostat: Last lecture.
- Program for detecting the type of a triangle: This lecture.

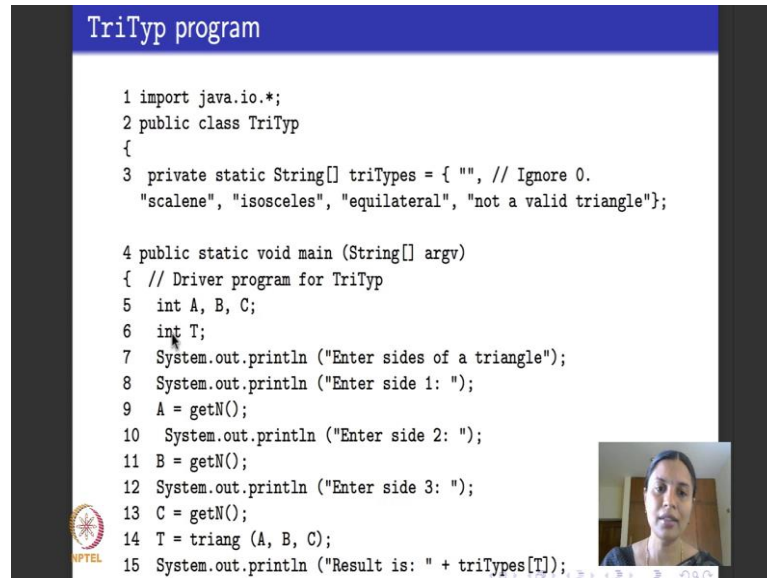
NPTEL

The slide features a blue header with the title 'Two examples'. Below the header, the text 'We look at two examples to understand reachability for logical predicates involving internal variables.' is followed by a bulleted list of two programs. The first bullet point is 'Program for a thermostat: Last lecture.' and the second is 'Program for detecting the type of a triangle: This lecture.' In the bottom right corner, there is a small video inset showing a woman speaking. The NPTEL logo is located in the bottom left corner of the slide area.

And to be able to understand the problems related to reachability and controllability well, I will walk you through 2 examples that illustrate the use of logical coverage criteria over source code. In the last lecture, we did the example of thermostat. There was one large predicate with the 4 clauses in that example. We showed how to solve for that predicate, that predicate had the clause with an internal variable. How to reach that predicate how to solve for it, and how to write test cases for predicate coverage. We also saw how to write test cases for code related active clause coverage for that predicate.

In today's lecture, we will take another program which is a very popular program that you will find in a few other text books in software and testing to test for decision coverage or logic coverage.

(Refer Slide Time: 01:27)



```
TriTyp program

1 import java.io.*;
2 public class TriTyp
3 {
4     private static String[] triTypes = { "", // Ignore 0.
5         "scalene", "isosceles", "equilateral", "not a valid triangle"};

6     public static void main (String[] argv)
7     { // Driver program for TriTyp
8         int A, B, C;
9         int T;
10        System.out.println ("Enter sides of a triangle");
11        System.out.println ("Enter side 1: ");
12        A = getN();
13        System.out.println ("Enter side 2: ");
14        B = getN();
15        System.out.println ("Enter side 3: ");
16        C = getN();
17        T = triang (A, B, C);
18        System.out.println ("Result is: " + triTypes[T]);
19    }
20 }
```

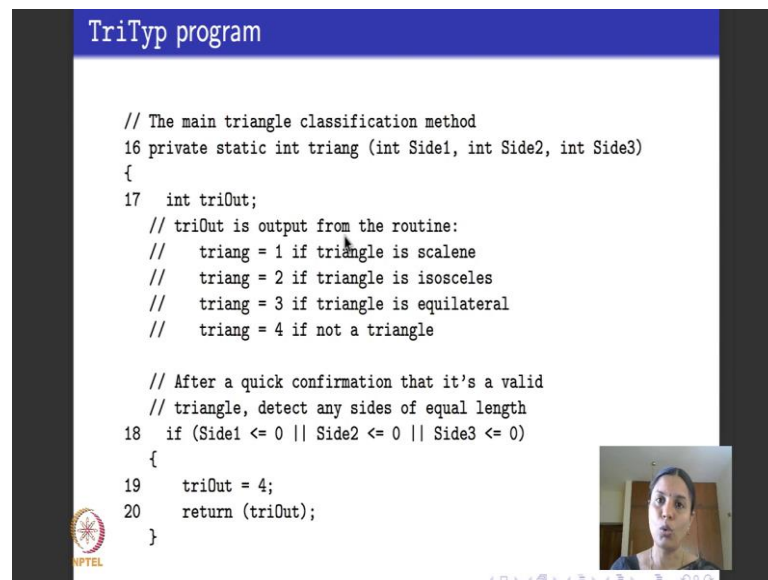
And I will tell you how to apply the predicate coverage criteria that we have learnt to be able to do testing for that program. That program is about detecting the types of the triangle. As again this is a program that I have taken from the textbook first edition of Paul Ammann and Jeff Offutt, the authors wrote this program. In other books you might find the control flow graph corresponding to a program and this program basically is called TriTyp, in short for type of a triangle and it detects these 3 types of triangle or it says that it is not a valid triangle.

So, in the triangle could be scalene triangle in which case all the 3 sides are not are of different length in a scalene triangle. In an isosceles triangle 2 of the 3 sides have the same length. In an equilateral triangle all the 3 sides have a same length. Lets say one of the sides that is entered is a negative number then this program will output say in that the triangle that you are trying to give us input in 3 sides is not a valid triangle.

Here is the program. So, what it does is that it has 3 integer variables A, B and C represented to be taken as inputs for the 3 sides of a triangle and then it has an integer T we will see what this does. T is for giving the output T returns the output which is a type of triangle. So, the 3 sides of a triangle that taken as input here. So, the first side is taken as input in A, the second side is taken as a input in B, the third side is taken as a input in C and then this main program calls this method called triang with inputs A, B and C as the 3 sides. This triang runs its code. I have given you the code for triang in the next few

slides and returns a number to be stored in T and then the system outputs saying the result is T. So, T is a number that represents one of these 3 types of a triangle or T could be a number that says that it is not a valid triangle. So, here is the main triangle classification method.

(Refer Slide Time: 03:27)



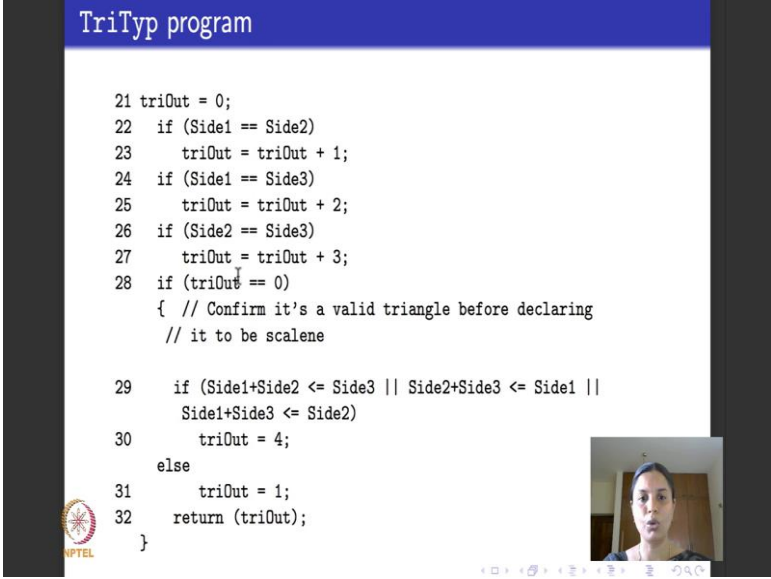
This one, triang. As I told you code will be presented over several slides please read it as a contiguous fragment of code and from end to end the code may not be executable. We are concentrating only on the fragment as its relevant for us to be able to understand test case design.

So, this is the second slide containing a continuation of the same code. It has triang method. So, here is what it outputs? It has an internally variable within this method called triOut; triOut is the output from that method. So, if what does it output? It outputs one if the triangle is a scalene triangle, it outputs the number 2 if the triangle is isosceles, it outputs the number 3 if the triangle is equilateral, it outputs 4 if 3 sides constituted that were entered do not constitute a valid triangle. So, what we first do here in this method is we first check if the triangle is valid or not. Why will it not be a valid triangle ? It will not be a valid triangle if one of the sides is a negative length. So, that is what this if checks for. It says if side one is less than or equal to 0 or side 2 is less than or equal to 0 or side 3 is less than or equal to 0 then you set triOut to be 4. What us triOut to be 4 mean ? If you go above and see triOut 4 mean that it is not a triangle.

So, you set triOut to be 4 and return triOut. So, here basically after taking the 3 sides as inputs in A, B and C, they get passed to the triang method A, B and C. triang method takes them as inputs side1, side 2 and side 3. First thing that it checks is if any of the sides is less than or equal to 0 that is this if statement in line number 18. If any of the 3 sides is less than or equal to 0, it out sets triOut to 4 which we interpret in the code to mean as it is not a valid triangle, returns it and comes out now it. Now after this its ruled out the fact that it is not a valid invalid triangle.

From now on it is working with the valid triangle it just has to find its type. So, what are the 3 types that the code can output? The 3 types that the code can output are whether it is a scalene triangle, isosceles triangle or an equilateral triangle. If it is a scalene triangle it outputs one, if its isosceles it outputs 2, if its equilateral it outputs 3. This is how the code is written.

(Refer Slide Time: 05:50)



```
TriTyp program

21 triOut = 0;
22 if (Side1 == Side2)
23     triOut = triOut + 1;
24 if (Side1 == Side3)
25     triOut = triOut + 2;
26 if (Side2 == Side3)
27     triOut = triOut + 3;
28 if (triOut == 0)
29 { // Confirm it's a valid triangle before declaring
30     // it to be scalene
31
32     if (Side1+Side2 <= Side3 || Side2+Side3 <= Side1 ||
33         Side1+Side3 <= Side2)
34         triOut = 4;
35     else
36         triOut = 1;
37     return (triOut);
38 }
```

So, here is the rest of the code which determines which of the 3 types the triangle is and outputs the corresponding number. So, another thing that I would like to tell you is that the style in which this particular code is written is slightly long drawn and complicated. You could directly may be write a simpler code that will directly detect the type of the triangle to be one of the 3 types. By no means is this code shortest code, shortest length code and the simplest code to detect the type of a triangle. Its written in a slightly roundabout way with lots of predicates labeling various statements just to illustrate the

richness of the logic coverage. Remember we are using this example to understand logic coverage criteria better.

So, this code is written in a slightly long way. It started with lot of the predicates one after the other so that we could use it to understand logic coverage better, but it is a same thing, still correctly computes which is the type of the triangle. So, if it is not 4, which means if it is not a valid triangle, the code continuous here. It sets triOut to be 0 and then what it does it takes one side after the other and compares to see if its equal. So, let us say if side one is equal to side 2 it increment triOut makes it one, if side one is equal to side 3 it increments triOut makes it 2, if side 2 is equal to side 3 it increments triOut and adds 3.

So, by then it would have correctly computed if all 3 sides are equal, then the all 3 if conditions would have passed and the value of the variable triOut would be 3 correctly. If triOut is 0 then what it does here is it checks again. So, it says if side one plus side 2 is less than or equal to side 3, side 2 plus side 3 is less than or equal to side one, or side one plus side 3 is less than or equal 2 sides 2, there is a problem right? It cannot be the case because this violates the property of a triangle. So, it out, but saying it is not a valid triangle else it says yes it is a scalene triangle. So, this comment says that basically confirms that it is a valid triangle before declaring it to be scalene if triOut is 0 and it returns once it computes.



Now in this part it is trying to compute wether trying to figure out whether triangle is isosceles or scalene.

(Refer Slide Time: 08:02)

```
TriTyp program

// Confirm it's a valid triangle before declaring
// it to be isosceles or equilateral



33 if (triOut > 3)
34     triOut = 3;
35 else if (triOut == 1 && Side1+Side2 > Side3)
36     triOut = 2;
37 else if (triOut == 2 && Side1+Side3 > Side2)
38     triOut = 2;
39 else if (triOut == 3 && Side2+Side3 > Side1)
40     triOut = 2;
41 else
42     triOut = 4;
43 return (triOut);
} // end Triang
```



So, let say triOut value turns out to be greater than 3. It says if is equal to 3 then you directly set it to be 3 an output. Otherwise now you check if its equilateral or not which means you check if sum of 2 sides is greater than one side. These are the 3 ways in which sum of the 2 of the sides could be greater than the third side those are the 3 conditions here and the corresponding values of triOut are given here. In all the cases it sets triOut to be 2 because that is what it has to output to if the triangle is scalene and it returns that. Otherwise it says if any of these conditions are violated then it says is not a valid triangle and returns it. So, that brings us to an end of the code.

(Refer Slide Time: 08:51)

- The program has three inputs— read into variables A, B and C in the main program method and then passed to the formal parameters Side1, Side2 and Side3 in Triang().
- Solve for the internal variable triOut.
- Solve for reachability of the various predicates and then work with criteria.





So, in summary what is this program have a program has 3 input variables a B and C in the main program method. The 3 inputs are passed as formal parameters to the triangle method as side one, side 2 and side 3. Triangle method has an internal variable called triOut which is set to the value T to be returned back to the main method. So all these predicates, if you see there are so many predicates this triang method: the first predicate that you encounter is here at line number 18 which is this. and then we have 3 more predicates here are at lines 22, 24, 26 and 28, fourth one each of these 4 predicates have exactly one clause each. In line number 29 there is one more predicate with 3 clauses and here again in lines 33, 35, 37 and 39, there are 4 more predicates.

(Refer Slide Time: 09:47)

Predicates in TriTyp

- Line 18: (Side1 <= 0 || Side2 <= 0 || Side3 <= 0).
- Line 22: (Side1 == Side2)
- Line 24: (Side2 == Side3)
- Line 26: (Side3 == Side1)
- Line 28: (triOut == 0)
- Line 29: (Side1+Side2 <= Side3 || Side2+Side3 <= Side1 || Side1+Side3 <= Side2)
- Line 33: (triOut > 0)
- Line 35: (triOut == 1 && Side1+Side2 > Side3)
- Line 37: (triOut == 2 && Side1+Side3 > Side2)
- Line 39: (triOut == 3 && Side2+Side3 > Side1)

So, here is where we have listed all the predicates as they come in the code. At line 18 there was this predicate we checked whether the triangle was valid or not and then this was the part where it starts building towards checking if the triangle is isosceles or equilateral and if it versus scalene triangle it outputs this. Now it does to check if it is a valid triangle in line 29 and then these are the parts where it checks whether it is a isosceles or equilateral triangle.

So, this line what I have done is we have gone through the code here, pulled out all the predicates from the code with their numbers I just listed it here. So, if you are, you find it a little fast you could move back and forth between the code and this and check if the

predicate and a corresponding to each line has been currently listed in this code. After this what we want to do?

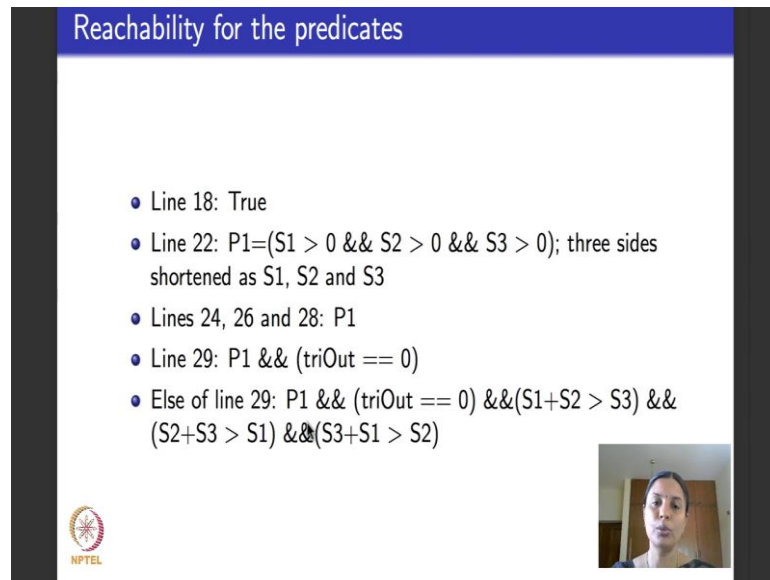
So, this is the set of full predicates that we want to work with. So, you can be ambitious and you can say that I will attempt each of the logic coverage criteria that we learnt for each of the predicates here. You could do that. If you try to do that you will realize that for these predicates to the predicates in line 22, 24, 26, 28 and the one line 33, how many clauses do each of these predicates have? They have only one clause right because they have no Boolean operators in between them and each of these predicates, the whole predicate is just one clause.

So, if you remember for predicates with one clause just predicate coverage is enough because you really cannot do anything much. For other kinds of coverage criteria you need more than one clause of the predicate. So, for each of these predicates you could directly write test cases that will test the predicate to be true once and false once. In fact, please try and do that as a small exercise for these predicates, the predicates in lines 22, 24 and 26, because you just have to give 3 values of sides such that here. For 22 side, one is the same as side 2 and for 24, side 2 is the same as side 3 and for 26, side 3 is the same as side one and then you are done. But for predicates in lines 28 and 33 you have to be able to give a value of triOut. Where does triOut come ? Let us go back and look at the code. triOut is an internal variable that belongs to this method triang which is the main triangle classification method.

So, these predicates in lines 28 and 33 which have only one clause which deals with triOut deal with an internal variable. But what are main inputs to the program? Three sides of a triangle. So, to be able to test this predicate let say triOut is equal to 0, I need to be able to give values for the 3 sides such that triOut is equal to 0 becomes true when the predicate should be true and I should be able to give values for the 3 sides, side one, side 2 and side 3 such that triOut is a value not equal to 0 for this predicate to be false.

Similarly, here for line 33, I should give values for inputs sides one, 2 and 3 such that triOut becomes any value greater than 0 which means it could take 1, 2, 3 or 4, it could even be an invalid triangle. To make this predicate false I must make triOut to be precisely equal to 0 which means I have to be able to give values in terms of inputs which are sides 1, 2 and 3.


(Refer Slide Time: 13:18)



Reachability for the predicates

- Line 18: True
- Line 22: $P1 = (S1 > 0 \ \&\& \ S2 > 0 \ \&\& \ S3 > 0)$; three sides shortened as S1, S2 and S3
- Lines 24, 26 and 28: P1
- Line 29: $P1 \ \&\& \ (triOut == 0)$
- Else of line 29: $P1 \ \&\& \ (triOut == 0) \ \&\& \ (S1 + S2 > S3) \ \&\& \ (S2 + S3 > S1) \ \&\& \ (S3 + S1 > S2)$

NPTEL



So, what we have to first do is we have to be able to find out reachability of the various predicates. So, from these slides onwards, so, this is the exhaustive list of all the predicates in the program. From these slides onwards I have illustrated this subsequent concept reachability, solving for the internal variable, computing coverage criteria. I have illustrated it only for a subset of the predicates, mainly because it is a repeat exercise if I want to be able to do it for all the predicates and for you to be able to learn it will be good if you can work it out for one or 2 predicates on your own. In particular I have not done for these predicates, in lines 35, 37 and 39.

So, feel free to work them out as a small exercise that you could do for yourself. So, what is the reachability of predicate in line 18? The predicate in line 18, if you go back to the program, happens to be the first predicate. So, it will always be reached because that is right the first statement in the method. So, the reachability for that is true means it will always be reached there is nothing else you need to do.

Now let us look at the next predicate. Predicate in line 22. As I told you to be able to reach here you have to be able to give any values side one, side 2, side 3 such that the predicate in line 18 skipped this if statement does not come and does not exist the method from here. If you do not want the method to exit from here you should be able to give values for side one, side 2 and side 3 such that this if condition is false. Once that becomes false you come here and you start executing the other predicates. That is what is


written here. It says in line 22 for we to be able to reach I must give side one to be greater than 0, I have abbreviated as side one S I D E one as S1 just to make it fit neatly into a slide. Similarly side 2 is abbreviated as S2, side 3 is also abbreviated as S3 and this whole thing we have gone ahead and called it as P 1 this is a new notation that I am introducing now so, that I could write P 1 here for reachability of predicates in lines 24, 26 and 28.

So, lines 24, 26 and 28 each of these predicates can be reached provided the values of all the 3 sides of the triangle is greater than 0 that is what it says. Now for line 29, I should get values of all the 3 sides greater than 0 and in addition the internal variable triOut must be equal to 0. Now line 29 was an if part with an else part which I dint give a line number for. Let us go back to the program. if you see this if part has this else part which I dint give a line number for. So, the reachability for that is the negation of the reachability for line predicate in line one.

(Refer Slide Time: 15:58)

Reachability for the predicates

- Line 33: $P1 \ \&\& \ (triOut \neq 0)$
- Line 35: $P1 \ \&\& \ (triOut \neq 0) \ \&\& \ (triOut \leq 3)$
- Line 37: $P1 \ \&\& \ (triOut \neq 0) \ \&\& \ (triOut \leq 3) \ \&\& \ ((triOut \neq 1) \ || \ (S1+S2 \leq S3))$
- Line 39: $P1 \ \&\& \ (triOut \neq 0) \ \&\& \ (triOut \leq 3) \ \&\& \ ((triOut \neq 1) \ || \ (S1+S2 \leq S3)) \ \&\& \ ((triOut \neq 2) \ || \ (S1+S3 \leq S2))$
- Else of line 39: $P1 \ \&\& \ (triOut \neq 0) \ \&\& \ (triOut \leq 3) \ \&\& \ ((triOut \neq 1) \ || \ (S1+S2 \leq S3)) \ \&\& \ ((triOut \neq 2) \ || \ (S1+S3 \leq S2)) \ \&\& \ ((triOut \neq 3) \ || \ (S2+S3 \leq S1))$



So, which means that it is this and these things are true. So, now, I continue and list reachability conditions for each of the other predicates. In each of these conditions I must be able to skip pass line number 18 which is P 1 and then write appropriate conditions. If you see, all this appropriate conditions deal with an internal variable triOut and there is a condition on triOut. Condition this one says triOut should not be equal to 0, this one says

triOut should be less than or equal to 3, this one says triOut should not be equal to 3, this one says triOut should not be equal to 1 and so on.

So, we need to be able to solve each of these predicates for the internal variable triOut. In particular we need to be able to give values for the 3 sides A, B, C or 1, 2, 3 such that here, in line 33, triOut turns out to be not equal to 0. In line 37 triOut turns out to be less than or equal to 3 and so on.

(Refer Slide Time: 16:53)



Solving for internal variable triOut	
triOut	Rules for determining triOut
0	$S1 \neq S2 \ \&\& \ S1 \neq S3 \ \&\& \ S2 \neq S3$
1	$S1 == S2 \ \&\& \ S1 \neq S3 \ \&\& \ S2 \neq S3$
2	$S1 \neq S2 \ \&\& \ S1 == S3 \ \&\& \ S2 \neq S3$
3	$S1 \neq S2 \ \&\& \ S1 \neq S3 \ \&\& \ S2 == S3$
4	$S1 == S2 \ \&\& \ S1 \neq S3 \ \&\& \ S2 == S3$
5	$S1 \neq S2 \ \&\& \ S1 == S3 \ \&\& \ S2 == S3$
6	$S1 == S2 \ \&\& \ S1 \neq S3 \ \&\& \ S2 == S3$

So, how do I solve? This is the table that tells you how to solve for the 3 predicates. So, if you go back look at the program and see when will the variable triOut be, take the values 0. It will take the value 0 if all the 3 sides are not equal to each other. So, that is what is this condition say triOut will be 0 if side one is not equal to side 2 and side one is not equal to side 3 and side 2 is not equal to side.

(Refer Slide Time: 17:34)

Reachability for predicates, after solving for triOut

- Line 29: $P1 \ \&\& \ (S1 \neq S2 \ \&\& \ S1 \neq S3 \ \&\& \ S2 \neq S3)$
- Else of line 29: $P1 \ \&\& \ (S1 \neq S2 \ \&\& \ S1 \neq S3 \ \&\& \ S2 \neq S3) \ \&\& \ (S1+S2 > S3) \ \&\& \ (S2+S3 > S1) \ \&\& \ (S3+S1 > S2)$
- Line 33: $P1 \ \&\& \ (S1 == S2 \ \&\& \ S1 == S3 \ \&\& \ S2 == S3)$
- And so on... keep substituting for triOut predicates.



So, similarly triOut will be one if one side a one set of sides are equal. Let us say side one is equal to side 2 and the other 2 are different similarly for 2, 3, 4, 5 and 6 right. These are the values that triOut will take. Now what I do is I go back and take this predicate in line number 29 it says P 1 and triOut is equal to 0. So, for triOut 0 to be 0 this is the condition that I have written out, this is the condition for reachability and controllability for triOut to be 0.

So, I will take that and substitute it back here. Now what I have achieved? I have achieved in rewriting predicate that comes in line number 29 purely in terms of inputs to the program. The predicate P 1 was in terms of the 3 sides and the fact the triOut was 0, I have removed that and rewritten it as another predicate that exactly will map to triOut being 0.

So, this is the predicate that I take and test for predicate coverage or clause coverage or correlated active clause coverage or whatever coverage criteria that I want to apply, whatever logic coverage criteria that I want to apply this is the predicate that I will test it for. What I have achieved in this predicate? what I have achieved is that I have achieved in completely eliminating the internal variable triOut from this program. I have rewritten the predicate to be purely in terms of inputs and outputs.

So, whatever coverage criteria I want to be able to do this in predicate now it becomes easier for me because I can directly give values to the 3 sides such that each of this

clauses have true false values to achieve my desired coverage criteria. Similarly the second item here, for the else statement of line 29, I have gone ahead and replaced the value of triOut to be 0 with this rule in the table. So, I have directly gone ahead and substituted, that gives me the long looking predicate and I had go on doing this for each of this predicates right, each of the predicates of in this side line 39, 37, 39 else 39 all of them have specific values for triOut.

So, I go on removing those triOut not equal to 0, triOut less than or equal to 3 and replace it with appropriate values and finish. I have not given them, but you can complete that by substituting it. Now what I have done the set of predicates that I have here, when I complete that set fully is all the predicates in the program with no internal variable in it. The predicates directly talk about the conditions of 3 sides and what are the values that each of this conditions will represent. I am ready to apply all my logic coverage criteria on these predicates.

(Refer Slide Time: 20:02)

Predicate coverage for the predicates									
		True				False			
	Predicate	A	B	B	EO	A	B	C	EO
line 18	$S1 \leq 0 \vee S2 \leq 0 \vee S3 \leq 0$	0	0	0	4	1	1	1	3
line 22	$(S1 == S2)$	1	1	1	3	1	2	2	3
line 24	$(S1 == S3)$	1	1	1	3	1	2	2	2
line 29	$(S1+S2 \leq S3 \vee S2+S3 \leq S1 \vee S1+S3 \leq S2) \vee$	1	2	3	4	2	3	4	1

Similarly, test cases can be written for other predicates too.

Now, so, what are the various logic coverage criteria that I have to apply? So, for example, I have taken predicate in line 18 which was like this and this is the table that tests for predicate coverage for each of these predicates. I have given you again only a subset of the predicates, you can write test cases for other predicates also.

How do you read this table? This first column here gives you the line number in which the predicate comes in the program. The second column actually lists what the predicate

is and these represent test case values for predicate coverage. The first set of values here test when the predicate is true the second set of values here test when the predicate is false. So, you give inputs A, B and C, I am sorry this is small error here, read this B that I am pointing to C and then what does EO represent? EO represents expected output. So, it says if the inputs to the 3 sides of a triangle A, B and C are these 3 then the expected output should be 3 and if the expected output is 3, then predicate coverage is tested for being false.



So, similarly predicate coverage is tested for being true. All the 3 triangles have been put 0, the expected output is 4 which it means it is not a valid triangle. Similarly for predicate in line 22 which S 1 is equal to S 2, I have given some value here and the output here should be correctly, if you see all the 3 values here are the same, so output is an equilateral triangle. Here the output is false so it is only 2 sides are equal.

Similarly for here it is predicate is true again predicate is false correct because side one is different from side 3. Now in line 29 we had this predicate S 1 plus S 2 is less than or equal to S 3 or S 2 plus S 3 is less than or equal to S 1 or S 1 plus S 3 is less than or equal to S 2 and these are the test case values that will test for the predicate being true. This is the expected output and for the predicate being false these are the test case values and this is the expected output. So, here for some of the predicates in my triangle program, I have written test cases for achieving predicate coverage for both true and false.

(Refer Slide Time: 22:16)

Clause coverage for the predicates									
Clause coverage is considered only for the predicates having more than one clause.									
Clause	True				False				EO
	A	B	B	EO	A	B	C	EO	
line 18 (S1 ≤ 0)	0	1	1	4	1	1	1	3	
(S2 ≤ 0)	1	0	1	4	1	1	1	3	
(S3 ≤ 0)	1	1	0	4	1	1	1	3	
line 29 (S1+S2 ≤ S3)	2	3	6	4	2	3	4	1	
(S2+S3 ≤ S1)	6	2	3	4	2	3	4	1	
(S1+S3 ≤ S2)	2	6	3	4	2	3	4	1	

Similarly, test cases can be written for clause coverage for other predicates too.



So, similarly we can do clause coverage. We didn't do clause coverage for the thermostat example in the last lecture. That predicate also had 4 clauses there. Just for illustrative purposes I have done clause coverage here for some of the predicates. Please remember one point that we discussed little while ago in this lecture. If a predicate has only one clause then that does not make sense to do clause coverage. So, a predicate should have more than one clause to be able to do clause coverage.

So, I have taken 2 example predicates that have more than one clause. The first one is the predicate in line number 18 that had 3 clauses: S1 less than or equal to 0 first clause, S2 less than or equal to 0 second clause, S3 less than or equal to 0 third clause. Again in the same way, read this whole thing. These are 3 sides of the triangle, expected output for the clause coverage to be true this clause to be true. These are 3 sides of the triangle, expected output for clause to be false.

Similarly, for the predicate in line number 29 there are 3 clauses and these are the test cases that will test for each clause to be true in turn and each clause to be false in turn. Similarly for all other predicates that have more than one clause you could go ahead and write clause coverage.

(Refer Slide Time: 23:28)

CACC for the predicates							
	Predicate	Clauses			A	B	C EO
line 18	$S1 \leq 0 \vee S2 \leq 0 \vee S3 \leq 0$	T	f	f	0	1	1 4
		F	f	f	1	1	1 3
		f	T	f	1	0	1 4
line 29	$(S1+S2 \leq S3 \vee S2+S3 \leq S1 \vee S1+S3 \leq S2) \vee$	T	f	f	2	3	6 4
		F	f	f	2	3	4 1
		f	T	f	6	2	3 4
		f	f	T	2	6	3 4

Similarly, test cases can be written for CACC for other predicates too.

Now, I have attempted correlated active clause coverage for the predicates. Here again to be able to do, consider a predicate like this in line 18: $S1 \leq 0 \vee S2 \leq 0 \vee S3 \leq 0$. It has 3 clauses: you have to make each clause the major clause in turn make it determine the predicate. That is making, we are assuming that this is a compute p subscript a and assuming that this is B, compute p b and assuming that this is c compute p c. Then you populate a table that looks like this which makes the first clause true and false second clause true and false then you have to eliminate duplicate rows.

And then finally, you will get this set of true false values. I have skipped all those steps, but I advise you to strongly try and do that it will be a good exercise and revising how to make each clause a major clause and make it determine the predicate and prune the truth table by removing the rows that occur as repetitions. Once you do that you will realize that you will be reduced with just these 3 rows: the first row makes clause a true; a is the major clause; clauses b and c false. Here is the set of test case for that, here is the expected output. Second row makes clause a the major clause this time its false b and b are false. Again here are the inputs, here is the expected outputs.

Similarly, for this exercise these predicate also we have done this. So, hopefully this second example would have helped you to understand how to do various logic coverage criteria for the predicates in the fairly large program this program that detected the type

of triangle was a fairly large program, several predicates. Feel free to write to me on the forum, if you want a have any doubts or if you are not able to complete some of the coverage criteria. Try them out and write to me will be happy to answer any questions.

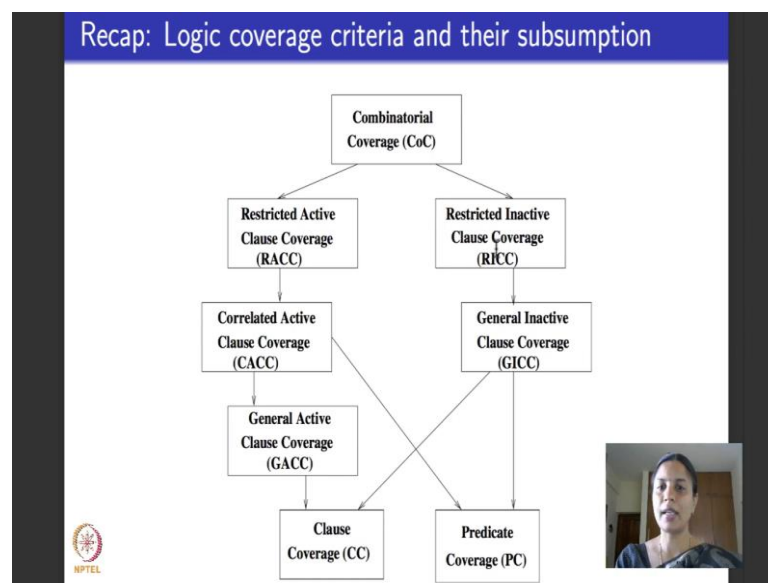
Thank you.

Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 27
Logic Coverage Criteria: Issues in applying to test code

Hello again. Welcome to the third lecture of the sixth week, we are in logic coverage criteria. Last two classes, I told you about how logic coverage applies to source code.

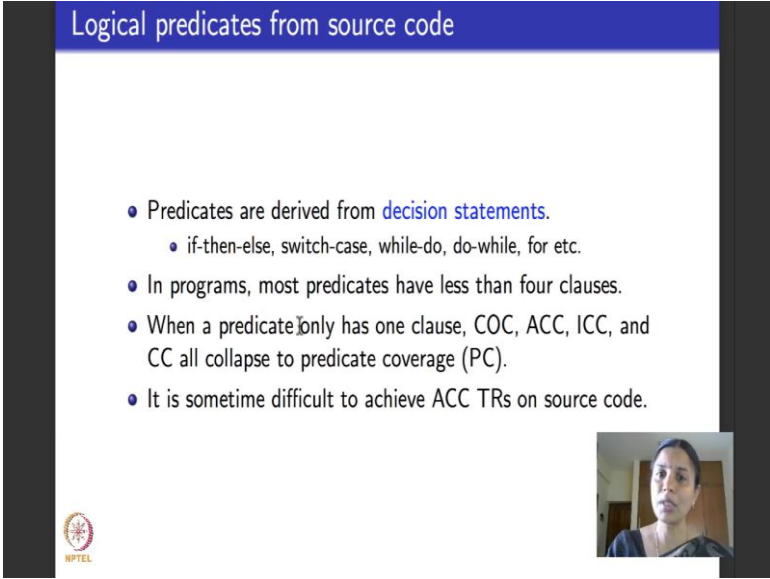
(Refer Slide Time: 00:26)



This is a recap slide. We have seen so many logic coverage criteria and using examples of 2 kinds of programs thermostat and detecting the type of a triangle. Using those example we saw how to practically apply predicate coverage, clause coverage and correlated active clause coverage. These 3 are considered to be the most useful amongst the various logic coverage criteria.



You would have hopefully realized through the examples that we had to do a good amount of work for even a fairly reasonably sized program like triangle type which had a good number of predicates to be able to reach the inner predicates, and solve for the internal variables, rewrite the predicates purely in terms of inputs. Then they looked very big and then the tables getting the tables for active clause coverage criteria was a bit of an effort.

(Refer Slide Time: 01:27)



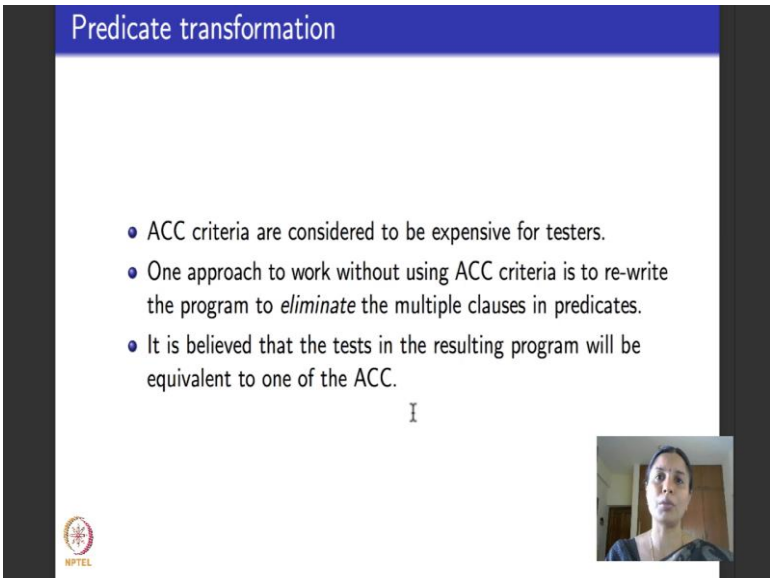
Logical predicates from source code

- Predicates are derived from **decision statements**.
 - if-then-else, switch-case, while-do, do-while, for etc.
- In programs, most predicates have less than four clauses.
- When a predicate only has one clause, COC, ACC, ICC, and CC all collapse to predicate coverage (PC).
- It is sometime difficult to achieve ACC TRs on source code.





So, imagine few thousand lines of code, which is what a standard program is all about and applying logic coverage criteria to that. It is going to be pretty non trivial. So, there is always been a bit of a debate, you know when predicates come from these kinds of statements and predicates have less than 4 clauses and if a predicate has more than one clause then, we said in the last class in the slide that active clause coverage criteria is very useful. But we saw through examples that its usually a little difficult task right. It needs a lot of human intervention and explicit knowledge about the code to be able to apply active clause coverage criteria and derive the test requirements for these criteria.

(Refer Slide Time: 01:59)



Predicate transformation

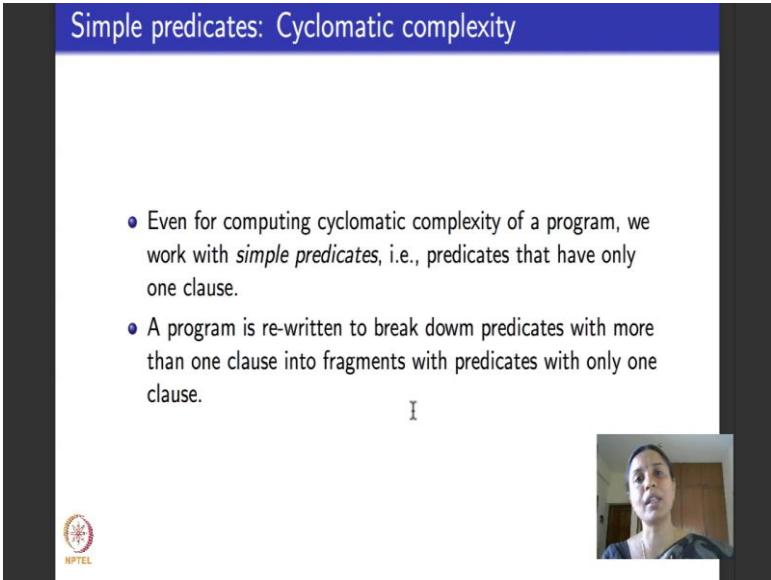
- ACC criteria are considered to be expensive for testers.
- One approach to work without using ACC criteria is to re-write the program to *eliminate* the multiple clauses in predicates.
- It is believed that the tests in the resulting program will be equivalent to one of the ACC.



So, there is been a lot of debates in the programming community. In fact, there is been a fairly large school of thought which says that why do we consider active clause coverage criteria because predicates has more than one clause. At least in those 2 examples we saw that predicate had several clauses, sometimes up to 4 clauses and then solving for the variables in that clause substituting back to them, taking each clause in turn be a major clause, filling up the truth table for correlated active clause coverage took a bit of time. So, there is been this school of thought that why not to rewrite the program to eliminate these many clauses in the predicate. Rewrite it to make sure that each predicate has one clause. I will show you some examples of what it means to rewrite a program.

So, lot of people have been thinking. And in fact, at some point in time there was a conjunction going around that if you take a program which has a predicate with let say 2, 3 or 4 clauses then using the semantics of operators like and nor which connect the clauses you will be able to rewrite the program by breaking the if statements into smaller statements. And it was believed that the resulting program, if I apply predicate coverage it will be the same as applying active clause coverage in the given program. Very soon it was proved that this is not correct.

(Refer Slide Time: 03:38)



Simple predicates: Cyclomatic complexity

- Even for computing cyclomatic complexity of a program, we work with *simple predicates*, i.e., predicates that have only one clause.
- A program is re-written to break down predicates with more than one clause into fragments with predicates with only one clause.

I

NPTEL

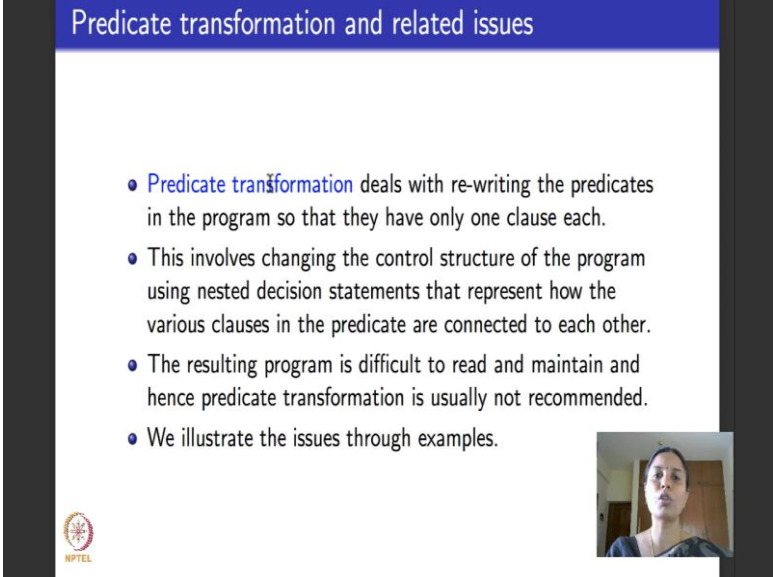
The slide features a blue header with the title 'Simple predicates: Cyclomatic complexity'. Below the header, there are two bullet points. The first bullet point states that for computing cyclomatic complexity, simple predicates (those with only one clause) are used. The second bullet point states that a program is rewritten to break down predicates with more than one clause into fragments with predicates that have only one clause. A small video inset in the bottom right corner shows a person speaking. The NPTEL logo is in the bottom left corner.

I will show you a examples that illustrates that predicate coverage in the transformed program is not the same as the desired active clause coverage. It could be correlated, it could be general either cases it may not be the same, we will see it through examples.

Another reason why people believed that predicate transformation is necessary is, remember I told you what how to do source code classical testing for source code write in the third week of or something like that. So, there we saw this notion of cyclomatic complexity which determined the number of independent paths in the program. One of the basic conditions in the cyclomatic complexity is when I consider control flow graph of the program all the predicates in that should be what are called simple predicates which means now, in terms of logic coverage criteria that we have seen, they should have exactly one clause.

So, it was recommended that you rewrite the program to be able to do that. So, there are lots of schools of thoughts which keep recommending that if there is a predicate which has 3 or 4 clauses make it simpler, rewrite it. What I want to do in today's lecture is using examples show you that it may not always be a good idea to rewrite the program to break down the several clauses in the predicate. It usually does not work all the time. We will see 2 examples, try to rewrite the predicate which come in this examples and see what are the issues that we will get, right.

(Refer Slide Time: 04:48)



Predicate transformation and related issues

- Predicate transformation deals with re-writing the predicates in the program so that they have only one clause each.
- This involves changing the control structure of the program using nested decision statements that represent how the various clauses in the predicate are connected to each other.
- The resulting program is difficult to read and maintain and hence predicate transformation is usually not recommended.
- We illustrate the issues through examples.

NPTEL

A small video inset in the bottom right corner shows a person speaking.

So, what is the notion of predicate transformation ? The process of predicate transformation deals with rewriting the predicates in the program so that they have, in the resulting transformed program each predicate has only one clause. This obviously involves change in the control structure of the program. Like for example, if I have if a

and b which has 2 clauses a and b, I break it up I have to put nested statements. First test for if a: if a is true then you test for if b: if b is true then it means both a and b are true then you write the then part of the original predicate here.

So it obviously means adding more control structure to the program. Lot of nested if statements lot of else statements right. The resulting program becomes very difficult to read. It has too many if statements too many nested if else statements then you do not know what is happening where, what is going where and usually because of this people do not like predicator transformation. Also programs large programs are written and supposed to be maintained for several years. So, for maintainability which means a third programmer or a developer or a maintenance engineer who is going to be able to look at your program and analyze it we will find it very cumbersome if you had written these complicated nested if then else to be able to just simplify for the sake of avoiding predicates with multiple clauses. It does not work, I will show you why using 2 examples.



(Refer Slide Time: 06:14)

Predicate transformation: Example #1

Consider the following program segment where a and b are Boolean clauses and S1 and S2 are single statements, a block of statements or function calls.

```
if (a && b)
    S1;
else
    S2;
```

TR for CACC criterion for the predicate $a \wedge b$: $\{(t,t), (t,f), (f,t)\}$.



Here is a small program fragment, it is only a program fragment not the entire program, meant to illustrate issues with predicate transformation.

So, this program fragment has 2 clauses a and b there is a predicate here in the if statement which says if a and b then you do statement or set of statements S1. S1 could be any statement, single statement, a block of statements, could have if again inside that,

could have other decision statements like while could have function calls. We do not care about it we just abstract it out call it S1. So, this clause says if a and b do S1, otherwise you do S2. This is just a small fragment. Now the predicate under consideration is this one a and b, as you can see it has 2 clauses.

So, I want to let say be make apply correlated active clause coverage criteria to this predicate I want to apply CACC criteria for this predicate. Now what you have to do? You have to do the same old exercise a be the major clause work out pa that is when a determines p, and then consider b to be the major clause workout p b which says when b determines p put them together to obtain correlated active clause coverage criteria where a becomes true, false once b becomes true, false once. We worked up the same example p is equal to a and b when we did this in the slides and inferring from those slides that we did for CACC for a and b here is the TR. I am not re doing it you can refer back to those slides.



The final test requirement for CACC for a and b will be this. How do you read this ? The first pair says make a and b true, second pair says make a true b false, third pair says make a false b true. Between these 3, correlated active clause coverage is completely satisfied for the predicate p which is a and b in our case. Now the goal is we will looking at program transformation this predicate has this 2 clauses. For some reason I do not like it, I do not want to do this CACC criteria, I want to break this 2 clauses and make rewrite this program into a program that has predicates that contain exactly one clause.

(Refer Slide Time: 08:35)

Predicate transformation: Example #1

Transformed program to remove the &&, functionally equivalent to the given program.

```
if (a)
{
    if(b)
        S1;
    else
        S2;
}
else
    S2;
```



So, here is an attempt at rewriting. This program, in this slide, if a and b do S1 else do S2 is rewritten like this. How do we understand what this is? So, if you read this program it says if a then you go here then you test for if b.

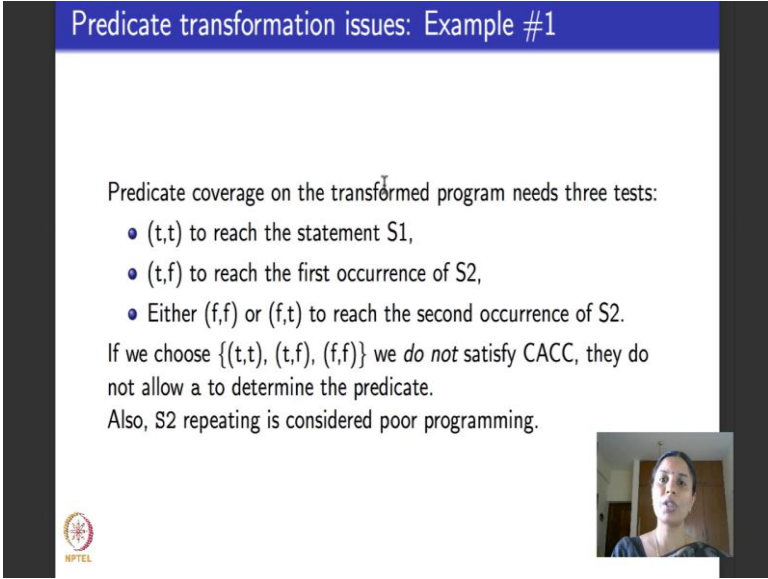
So, let say both these fs pass then what is it mean it means a is true and b is true or going back by the semantics of the given program a and b are true then what do I do? I do S1 it is a same thing that I do here. If a is true and then b is true I do S1 else, this else is for this if, which means a is true and b is false. So, a and b is false then I do S2. What does this one say? This one is the else for the first if here it says if a is false then you do S2. So, what I have achieved here, I have taken the program in this slide which says if a and b are both true do S1 else do S2 and I have broken it up like this I said if a is true and b is true do S1 else which means if a is true and b is false do S2. If a itself is false do not bother about checking b because a and b is going to be false directly do S2. So, this captures exactly the same semantics as this program, but it is rewritten to ensure that each predicate here has exactly one clauses.

So, there are 2 predicates one just has clause a one just has clause b. I have eliminated the predicates containing 2 clauses here and this is the new control structure that I have arrived at. Now this does not look too bad but typical experienced programmers will not like this control structure. There are several disadvantages to it. One is people do not like the fact that this entire code segment S2 gets repeated here. That is not considered good

programming practice at all and we cannot avoid it here, because we had our goal was to remove and that was there in the previous predicate and that process we ended up creating another copy of the code for S2. It is not consider a good programming practice.

And the other thing is now let us consider predicate coverage for this transformed program. Remember the conjecture was this is the transformed program, I have removed predicates with multiple clauses. If I achieve predicate coverage in this it will mean that achieving a appropriate active clause coverage in this original program. What we will show is that predicates coverage in this transformed program will not satisfy CACC criteria for the original program. What is predicate coverage for this transformed program? Predicate coverage for this transformed program will need three test cases: first one is a is true, b is true that will reach this statement S1.

(Refer Slide Time: 11:07)





Predicate transformation issues: Example #1

Predicate coverage on the transformed program needs three tests:

- (t,t) to reach the statement S1,
- (t,f) to reach the first occurrence of S2,
- Either (f,f) or (f,t) to reach the second occurrence of S2.

If we choose {(t,t), (t,f), (f,f)} we *do not* satisfy CACC, they do not allow a to determine the predicate.

Also, S2 repeating is considered poor programming.

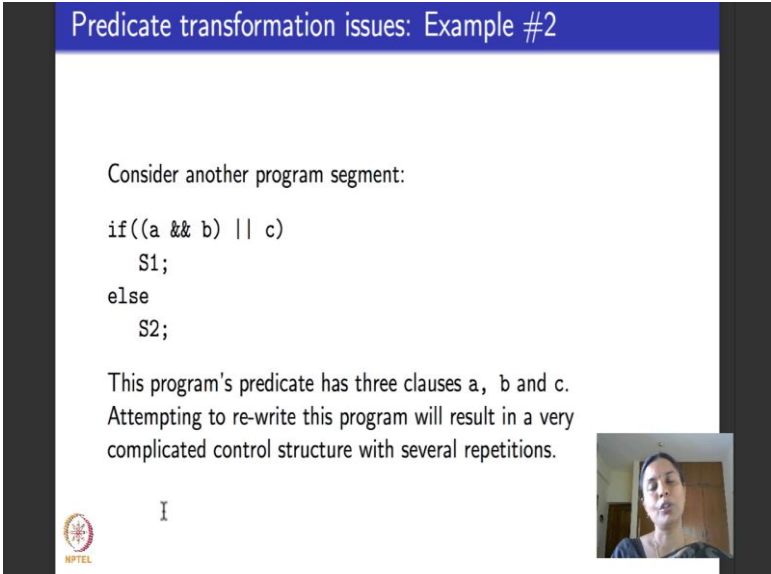


Second one is a is true, b is false, that will reach this statement S2. The third one which is a is false as I told you, but b could be either false or true which means a false and b false or a false and b true. Both these test cases will reach this S2. So, the first test case for predicate coverage is both a and b are true, reach S1. Second test case is a is true b is false, so, reach this S2. The third test case which needs to reach this copy of S2 can be achieved in 2 ways: when is both a and b could be false or a could be false and b could be true both of them will reach this second copy of S2.

So, that is what is given here. But let say I have a choice here in the third case right for either taking both of them to be false or a to be false and b to be true, but let us say I take both of them to be false. So, the final set of test cases that I choose for predicate coverage are: true, true to reach S1, true, false from the second item to reach S2, the first occurrence of S2 and from the third one I choose false, false, to reach this second occurrence of S2. So, this satisfies predicate coverage for the transformed program, but it does not satisfy CACC for the original program. CACC for the original program as we have discussed, had needed 3 test cases a true, b true; a true, b false and a false, b true. This was the only option left where as in predicate coverage I had the choice between choosing this or this and I decided to choose both to be false because of that I do not satisfy CACC, but I satisfy predicate coverage.

So, clearly the conjecture that you transform the program, breakup predicates with more than one clauses into predicates with single clause and try to do predicate coverage on the transformed program that will be equivalent to CACC or GACC on the given program that is not true for this example. Also the transformed program has this not so welcome feature of the code fragment S2 being copied ditto twice, that is usually not accepted.

(Refer Slide Time: 13:32)



Predicate transformation issues: Example #2

Consider another program segment:

```
if((a && b) || c)
    S1;
else
    S2;
```

This program's predicate has three clauses a, b and c. Attempting to re-write this program will result in a very complicated control structure with several repetitions.

I

NPTEL

The slide features a blue header with the title 'Predicate transformation issues: Example #2'. Below the header, the text 'Consider another program segment:' is followed by a code snippet for an if-else statement. The code snippet is: `if((a && b) || c)` followed by `S1;` on the next line, and `else` followed by `S2;` on the next line. Below the code, a paragraph states: 'This program's predicate has three clauses a, b and c. Attempting to re-write this program will result in a very complicated control structure with several repetitions.' At the bottom left of the slide is the NPTEL logo, and at the bottom center is the letter 'I'. A small video inset in the bottom right corner shows a person speaking.

So, I show you one more example to understand the same issue, but we will look at it from a slightly different prospective. Here is an example that has a predicate with 3

clauses that are connected like this: a and b or c with d. If a and b or c is true then you do S1 else you do S2, right. So, this is easy to read, if you notice because I just can read this directly any programmer will understand what is happening here. It says if a and b is true or c is true then you do S1. If it the case that both of them are false in which case the whole thing is false then you do S2. Now let us attempt to rewrite this program. You can attempt in several different ways. Whatever ways you attempt, we will realize that you will get a very long program with several occurrences of S1 and S2 repeated and it will have a very ugly looking control structure.

(Refer Slide Time: 14:26)

Predicate transformation issues: Example #2

Transformed program to split the three clauses.

```

if (a)
  if (b)
    if (c)
      S1;
    else
      S1;
  else
    if (c)
      S1;
    else
      S2;
else
  if (b)
    if (c)
      S1;
    else
      S2;
  else
    if (c)
      S1;
    else
      S2;

```

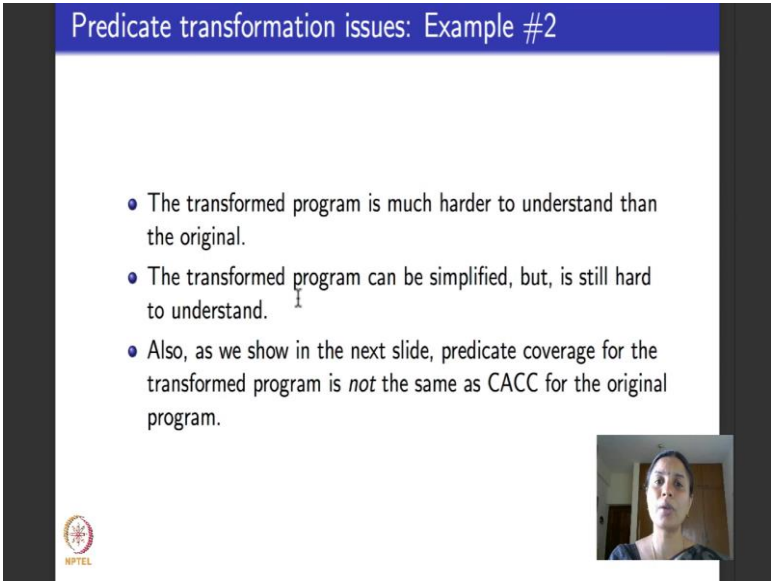
So, here is an attempt, one attempt at rewriting the program. So, if you look at this itself you will realize that this looks very complicated right and quite difficult to understand unlike this, this is neat. The only thing that you had here was you had a predicate with 3 clauses, but otherwise its quite neat and simple. For some reason you wanted to eliminate this predicate with 3 clauses, but you wanted to retain a new program where each predicate as a single clause with exactly the same meaning and here is an one attempt to rewrite it. How do I read this? I read this as follows I say if a is true then you check if b is true b is also true. Then you check c is true if which means if I am here if c becomes true, then what does it mean? It means all 3: a, b and c are true then what do you do you do S1. If c is not true, but a and b both true then you still do S1. That correctly fix to the semantics of this program right. If a and b are both true and c is either true or false then this whole thing will be true all the cases I do S1.

So, I have simulated till here. Now let us look at this part. This part has an else that corresponds to this b. So, we are in a situation where if a is true, but b is false which is this else then you can still do S1 if c is true because that is what the semantics of this predicate is. So, you check if c is true then you do S1. If c is false and b is false which means the whole predicate is going to be false then you have to do S2. So, then you S2, is that clear. So, now, let us go continue, we have not finished all the cases. So, now, let us look at this else, if you go straight up corresponds to this first if which checks if a is true.

So, it means if you reach this else it means that a is false. So, I check if b is still true and b will be false, but if c is true then the or statement of the predicate in this slide becomes true. So, I can still do S1, but if c is false then I do S2 and again I check if b is false if c is true then I do S1, but if c is also false then I do S2. After this long drawn effort I have finally achieved to rewrite the program to capture the exactly the same meaning as this predicate.

But look at this program. You would agree with me that it looks completely ugly and unreadable. Just because I did not want this predicate, I try to attempt a transformation and that resulted in this. By the way this is not the only transformed program. There are several other ways of transforming it I just wrote this longest transformation in some sense to illustrate how ugly a transformed program can get.


(Refer Slide Time: 17:06)



Predicate transformation issues: Example #2

- The transformed program is much harder to understand than the original.
- The transformed program can be simplified, but, is still hard to understand.
- Also, as we show in the next slide, predicate coverage for the transformed program is *not* the same as CACC for the original program.

NPTEL



But the point that I wanted to illustrate is that whatever is the transformed program its not unique, this is not the only transform program you could try to do it and you could probably get a simpler one. The transformed program is usually much harder to understand than the given one. You would agree with me because just by seeing his example and going back to our logical coverage criteria conjecture which said that predicate coverage on this transformed program should be the same as CACC on the given program, that is also false for this program as it would illustrate.



(Refer Slide Time: 17:39)

Predicate transformation issues: Example #2

Table comparing predicate coverage in the tranformed program with CACC in the original program.

	a	b	c	$((a \wedge b) \vee c)$	CACC	Predicate coverage
1	t	t	t	T		X
2	t	t	f	T	X	
3	t	f	t	T	X	X
4	t	f	f	F	X	X
5	f	t	t	T		X
6	f	t	f	F	X	
7	f	f	t	T		
8	f	f	f	F		X

I

So, here in this slide. I have given you details of correlated active clause coverage for the predicate which was this. I have written it in terms of logical notation 'and' and 'or', but it is the same predicate a and b or c that was here, a and b or c. Here its written in programming notation, here is written in logical notation, but it is a same predicate. This part the first five columns illustrates the truth table for the predicate. Predicate had 3 clauses a, b and c each clause can take true false values in turn and this is the entire truth table that assigns true false values to a, b and c and this is how the predicate evaluates it evaluates to true and all of them are true. It evaluates to false when b and c are false resulting in the whole thing being false it also evaluates to false when a and c are false or all 3 are false.

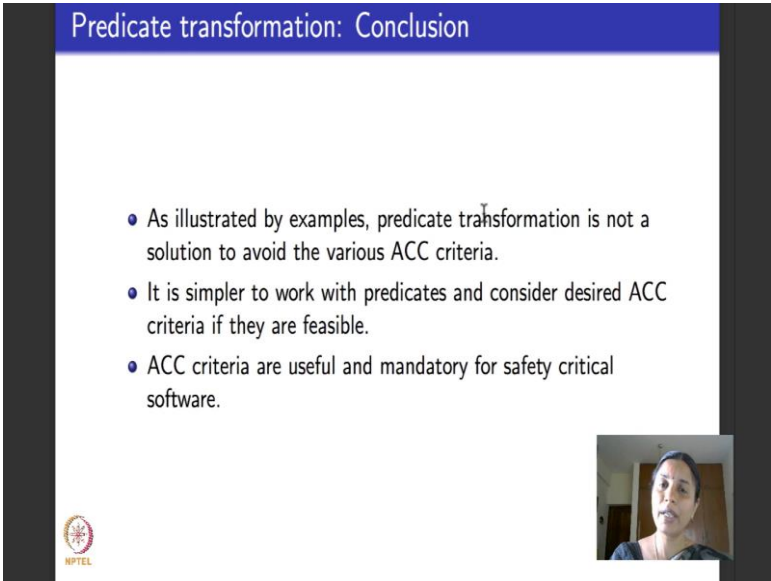
So, they tell you what the predicate evaluates to. Please do not get confused to this capital t and small t they both mean true. I have written it as capital t just to illustrate

what the resulting truth false values of the predicate is. Now if you see to achieve predicate coverage on the on the resulting program I can choose rows 1, 3, 4, 5 and 8 that will completely achieve predicate coverage. Why is that so, because if you see row 8 corresponds to predicate being false 5, 3 and 1 respond to predicate being true, row 4 also corresponds to predicate being false.

So, predicate coverage is achieved on the transformed program where each clause takes turns to be true and false resulting in the predicate to be true and false. But correlated active clause coverage on the original program, if you work out all the details by computing each clause to be the major clause do p a, p b, p c. Write out the truth table and do that exercise which I have not given here because we have done it for enough examples if you do that and do the final marking, the TRs for CACC will happen to be rows 2, 3, 4 and 6. So, if you compare the last 2 columns, you will realize there is immediately a miss match right, because row one satisfies predicate coverage, is not needed for correlated active clause coverage, row 2 satisfies correlated active clause coverage, but does not achieve dedicate coverage.

Similarly, row 6 satisfies correlated clause coverage does not achieve predicate coverage and row 5 satisfies predicate coverage, rows 5 and 8 satisfy predicate coverage, but are not needed for CACC. So, there is a lot of mismatch between these 2 columns which means that the TRs for CACC and for predicate coverage do not match for this example.


(Refer Slide Time: 20:26)



Predicate transformation: Conclusion

- As illustrated by examples, predicate transformation is not a solution to avoid the various ACC criteria.
- It is simpler to work with predicates and consider desired ACC criteria if they are feasible.
- ACC criteria are useful and mandatory for safety critical software.

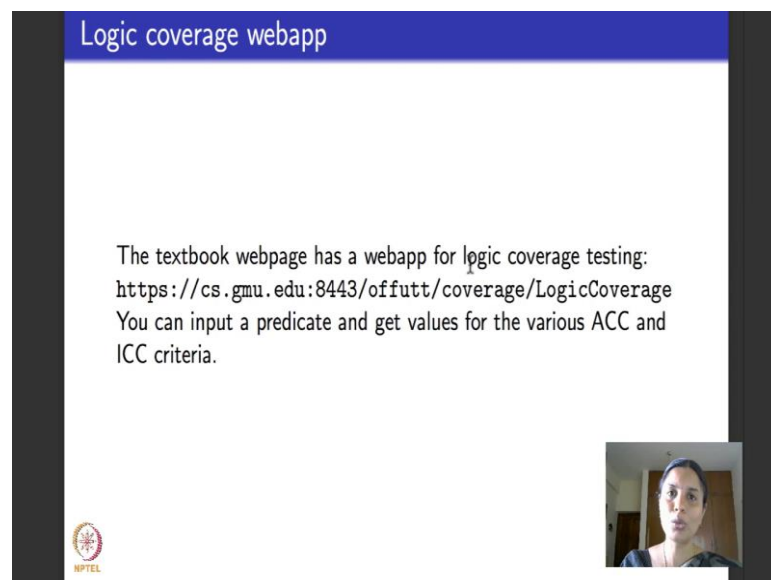
NPTEL



So, what I want you to illustrate to you using these 2 examples is that predicate transformation is not a good solution to avoid using active clause coverage criteria. Active clause coverage criteria is mandatory for testing safety critical software and has to be proven effective for finding lot of errors, especially for predicates that have 3 or 4 clauses.

So, it is a good idea to be able to go ahead and use it and transforming a program to remove those extra clauses in the predicate with the aim of avoiding active clause coverage criteria will not work. They are useful, they mandatory and it will be good to get used to it.

(Refer Slide Time: 21:08)



Logic coverage webapp

The textbook webpage has a webapp for logic coverage testing:
<https://cs.gmu.edu:8443/offutt/coverage/LogicCoverage>
You can input a predicate and get values for the various ACC and ICC criteria.

NPTel

In fact, to help you get some more practice, I encourage you to look at this book website. Most of the material as we have seen is derived from this text book called introduction to software testing. The book has a very good web page. By now you all should know the webpage. In that webpage there is a page that contains web apps. So, there is a web app for logic coverage criteria here is the URL for that app.

So, here you can input a predicate and get values for the various active clause coverage criteria and the various inactive clause coverage criteria. I do not think they do predicate in clause coverage because they are easier coverage criteria to meet, but they have this app that works well for ACC and for various ICC also. Try use it for a some predicate of your choice with 3 clauses or 4 clauses, work out the solution manually then use the app

to see if your answer matches with the answer given by the app. The app slows down a bit for predicate with larger clauses because they underlying problems that it tries to do to get these coverage criteria of each clause determining a predicate are non trivial problems.

So, try and use it for predicate with 3 or 4 clauses. That way you can work out the solution on your own and try and see if the solution that you have got is the same as the solution that the web app provides, and feel free to ask me if you have any doubts in the forum.

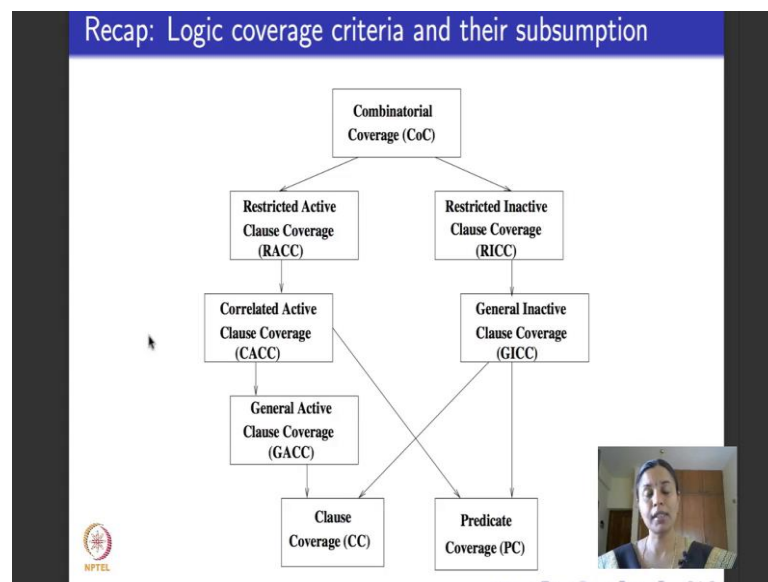
Thank you.

Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 28
Logic Coverage Criteria: Applied to test specifications

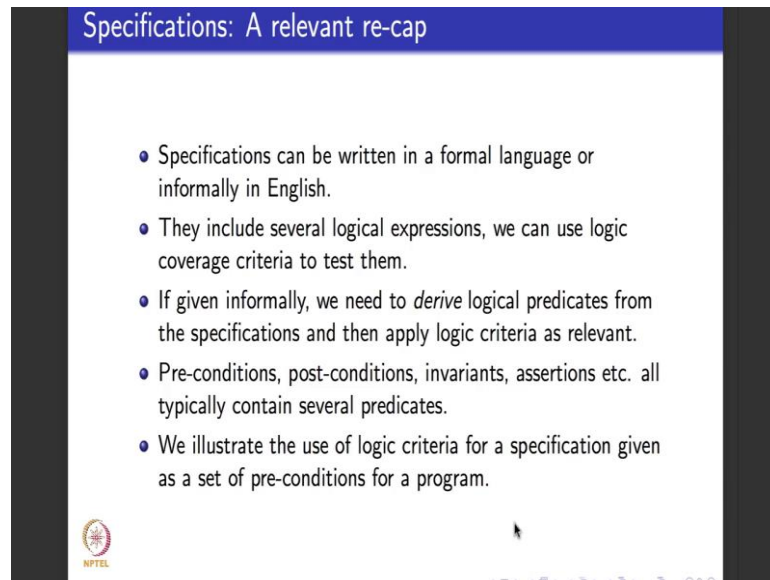
Hello everyone. We are in week 6 and today I am going to do the fourth lecture of week 6. As always we will continue with logic. What I will do today after we saw 2 lectures on how to apply logic coverage criteria for source code, for a change we will look at specifications, we will see what is the role that logical plays us for a specification is concerned.

(Refer Slide Time: 00:39)



And we will see how to apply the coverage criteria that we learnt over specifications. So, this is same slide I have been showing for the past few lectures just to help you recap logic coverage criteria. At the bases we have clause and predicate coverage, the whole predicate becomes true once, false once and predicate coverage. Each clause is made true once, false once and predicate coverage. Combinatorial coverage tests for the entire truth table. Then we have active clause coverage criteria, three of them. The most popular is correlated active clause coverage here in the center then we have inactive clause coverage criteria 2 of them, and when a predicate has only one clause all these coverage criteria basically boil down to just predicate coverage criteria.

(Refer Slide Time: 01:29)



Specifications: A relevant re-cap

- Specifications can be written in a formal language or informally in English.
- They include several logical expressions, we can use logic coverage criteria to test them.
- If given informally, we need to *derive* logical predicates from the specifications and then apply logic criteria as relevant.
- Pre-conditions, post-conditions, invariants, assertions etc. all typically contain several predicates.
- We illustrate the use of logic criteria for a specification given as a set of pre-conditions for a program.

NPTEL

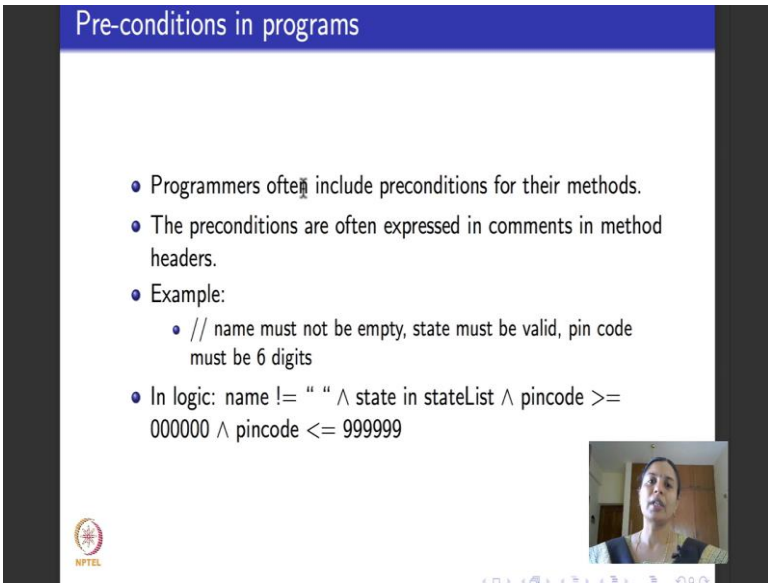
What we will do today is we look at specifications of requirements? It could be a part of requirement specification document, it could be a part of a design or an architecture document. We will see where do logic come into play in these kind of documents and how to apply the logical coverage criteria that we have learnt to generate test cases for specifications. So, typically where do specifications stay? You might have heard documents like software requirements specification or system requirement specifications they usually abbreviated as (SRS). Similarly you have functional requirements specifications (FRS) hardware requirement specifications (HRS). So, and these are always documents that are typically written in English. Somebody will write saying the server will respond to the client request within three milliseconds, the server will acknowledge every request by a client and so on. Or people use semi formal notations like use cases and so on too write these requirements.

Typically lot of these requirements include logical specifications like for example, they might write a specification which says that if the value of pin code is entered then it must be exactly a 6 digit number. So, this can be thought of as a formula or a condition that can be written in logic. Basically specifications involve conditions which talk about if this happens, this will happen, for this to happen something else must have happened. So, these conditions are basically nothing but logical predicates and if they are not given in the formal language of logic or in programming notation as testers is our duty to be able to read the English specification and derive logical predicates from them.

So, in today's example, we will look at English specifications and see how logical predicates can be derived from them and where do these kind of specifications come? These kind of specifications come typically as pre conditions, post conditions, invariants. Invariants, for example, it could be a property which says that at no point in time will if there is a critical region in a particular database or access to a database and in variant could say things like at no point in time 2 different processes have access to the critical regions. So, this can be thought of as an invariant describing the safety property of mutual exclusion.

Similarly, very common for programmers to write assertions to be able to debug their design or code. So you can find specifications there also. What we will do is we will take one of these examples, we will take either preconditions or post conditions. In this lecture I will take precondition as an example and I will show you how logic specifications are used to write these pre conditions and the coverage criteria that we learnt, how can we used to test that these pre conditions indeed hold.

(Refer Slide Time: 04:19)



The slide is titled "Pre-conditions in programs" in a blue header. It contains the following bullet points:

- Programmers often include preconditions for their methods.
- The preconditions are often expressed in comments in method headers.
- Example:
 - `// name must not be empty, state must be valid, pin code must be 6 digits`
- In logic: $\text{name} \neq "" \wedge \text{state} \in \text{stateList} \wedge \text{pincode} \geq 000000 \wedge \text{pincode} \leq 999999$

In the bottom right corner, there is a small video inset showing a person speaking. The NPTEL logo is visible in the bottom left corner of the slide.

So, where do preconditions occur in programs? Typically programmers include pre conditions for their methods in Java or if you writing in C you include pre conditions for your functions for the entire program before you begin the text of the program. Preconditions could be loosely documented they could just mean comments say you do this you do that and its assumed that the programmer would have written code that will

take care of the input satisfying these preconditions. These specific conditions on what the input look like, these specify conditions on what they output should be and so on. So, here is a small example. This is an example of a precondition. Lets say we are working with the program that deals with some kind of addresses. So, it says that in the address assuming that it is a record containing fields like name, street name, house number locality, state and pin code, you might want certain preconditions saying that the name should not be an empty stream, nobody should enter in address where the name is empty.

So, that is the first comment. The second comment says state must be valid. Like for example, if it is India then you cannot write any garbage name that you think as a state it must be one of this so many states that we have in India, it must be one of the valid states and lets say for an address the containing a pin-code in India, pin code cannot have a arbitrary number of digits pin code cannot just be written as a one digit number or as a 5 digit number, it should always be a 6 digit number. So, in any program that deals with a database that contains address, you might want to validate the data base by using the these three as example preconditions. They say name is not empty, state must be one of the predefined state because otherwise people might make spelling mistakes while entering.

So, you typically give a drop down menu where you let them choose the states, or pin and pin code must be exactly having 6 digits. So, typically they leave it at this. Somewhere in the code you will find parts that checks for these kind of things and give an exception when these things are not satisfied. But my goal is to be able to specify these preconditions as logical predicates and see if we can test them using the coverage criteria that we have learnt.

So, what do I write? I write it like this in logic I say name, as a variable name, should not be equal to, read this exclamation mark equal to as not equal to an empty string and state must be one in a list of states which I keep let us say in a list called state lists and pin code as a number should be greater than or equal to 0, as 6 different digits. You could even view it as a string because in some programming languages that could be treated as a number 0.

So, you will view it as a string containing 000000 and this order will be the lexicographic order and pin code should not be greater than or equal to this 999999. So, it is a number

that has exactly 6 digits or the string that has exactly 6 digits where the preceding 0s do matter.

(Refer Slide Time: 07:21)

Pre-conditions in Conjunctive Normal Form (CNF)

- A predicate is in **Conjunctive Normal Form (CNF)** if it consists of clauses or disjuncts connected by the conjunction (and) operator.
- Examples: $a \wedge b \wedge c$, $(a \vee b) \wedge (c \vee d)$.
- A major clause is made active by making all other clauses true.
- ACC TR is *all true* and then a *diagonal* of false values.

- Example:

	a	b	c
1	T	T	T
2	F	T	T
3	T	F	T
4	T	T	F
5	...		

When we talk about pre conditions in specifications, if you say this example that we saw what is the connecting \wedge operator in this precondition? If I view this as a predicate, I can say that this predicate has four clauses. This is the first clause which says name as non empty, this is the second clause which says state should be in the list of states, 3rd clause is pin code is greater or equal to this number fourth clause says pin code is less than or equal to this number. And these four clauses are connected by one operator which is the \wedge operator. Typically you will realize that most of these pre conditions, post conditions typically are always connected by an \wedge operator or by a \vee operator.

So, special kinds of predicates, there are normal forms which describe these kinds of predicates. So, a predicate is said to be in conjunctive normal form, the precondition that we saw here is in conjunctive normal form if it consists of clauses that are connected by an outermost conjunction operator or an \wedge operator.

In conjunctive normal form. these clauses could be disjuncts themselves. In the sense that one particular clause within itself could contain \vee operator. Of course, for going back out condition of logic coverage criteria we would distinguish its 2 different clauses, because for us a clause cannot contain an \vee operator, but in conjunctive normal form that is allowed. So, here are 2 examples of formulae and conjunctive normal form. The first

one $a \wedge b \wedge c$ has three disjuncts or clauses as we call it in this course: a , b and c and the outermost operator that connects these three disjuncts is an \wedge operator.

In this case, the second example I have $a \vee b \wedge c \vee d$ for us it has four clauses, but if you see the outermost operator is still an \wedge operator. Inside, the clauses could be connected by an \vee . So, if the clauses are connected by or this entire thing, $a \vee b$ in the jargon of logic, is what is called a disjunct. In the sense that these clauses individual clauses inside could be connected by a disjunction operator or an \vee operator, but the outermost operator connecting them is an \wedge . We also implicitly assume that inside there are no \wedge s.

For example, I should not have $a \wedge b \wedge c \wedge d$. If I have that then I say a is one clause disjunct and b is another disjunct I separate it out. The outer most operator in a conjunctive normal form should be an \wedge . So, it is a conjunct of disjuncts, that is how you call a conjunctive normal form formula as. Now, remember we want to be able to work with logic coverage criteria. Always it is easy to do predicate coverage and clause coverage, but to be able to do active clause coverage, if a predicate is in conjunctive normal form its fairly easy. What you will have to do is the following. Major clause that you want to focus and make the make it determine the predicate you make that active by making all other clauses true.

Let us take this example $a \wedge b \wedge c$. If suppose I want to make a as my major clause and I want a to determine the predicate p . So, I will make $b \wedge c$ completely true right, which are the minor clauses I will make them true. If b and c are true then the truth or falsity of the predicate is completely influenced by a , because when a becomes true, the predicate will be true and a becomes false the predicate will be false. b and c are true anyway so, they will not influence the predicate at all.

So, that is what this point. Here says it says major clause is made active, means active means made it made to determine the predicate by making all other clauses true. Then what is true truth what is a table containing the test requirements for active clause coverage look like? It look like this: it will have a row of all true's and then it will have a diagonal of false values. So, what do we mean by diagonal of false values? In this case I am making the major clause. So, a is true once false once the rest of the minor clauses are all true with that is what I wanted to be for a to be determining p . In this case I am making the major clause b determine p . So, b is false once true once and the rest of the

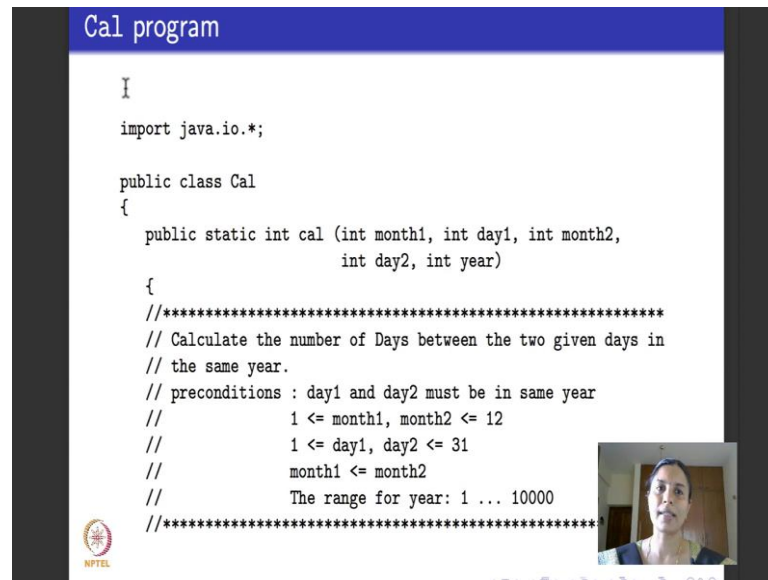
clauses will be true and so on. So, this is how ACC test requirement looks like when the predicate is given in conjunctive normal form. There is another popular normal form in logic called disjunctive normal form. It is the opposite of, somewhat the opposite of conjunctive normal form.

Here the outermost connective is an \vee . Inside various clauses could be connected by \wedge its reverse the role of \wedge and \vee are reversed in disjunctive normal form from conjunctive normal form. So, we say a predicate is in disjunctive normal form if it consists of clauses or conjuncts that are connected by a \vee operator. So, here are 2 examples: $a \vee b \vee c$ there are three clauses connected by an \vee operator. So, we say this predicate is in disjunctive normal form. Here again there are four clauses: $a \wedge b$ is one clause, $c \wedge d$ is one clause. This I mean sorry $a \wedge b$ is one conjunct, $c \wedge d$ is one conjunct, together there are four clauses. These conjuncts inside can have \wedge s, but in disjunctive normal form the outermost operator is an \vee .

So, when do you say a formula is in disjunctive normal form? We say a formula is in disjunctive normal form if it is an \vee of \wedge s. So for this to satisfy various logic coverage criteria in particular to satisfy active coverage criteria what do you do? You make the major clause active or determine predicate by making all other minor clauses false. So, in this example suppose you make a , b and c both false then the truth value a will completely influence the truth or falsity of the predicate. If a is false predicate will be false, if a is true because b and c are false the predicate will become true just because of a . So, here again for active clause coverage the TR is quite easy. It has a row of all false and then a diagonal of true values. True values go along the diagonal by making each clause take true in turn. The rest of the table is typically filled with false values and you would realize that in this case it nicely happens that each clause takes turns to be the major clause and does determine the predicate.

So, these two are two standard normal forms in which many preconditions and post conditions are written and when we do logic coverage criteria for these preconditions and post conditions, we will apply these truth tables to be able to determine ACC test requirements for them.

(Refer Slide Time: 14:21)



So, here is an example. This is an example of a calendar program, a simple program that calculates the number of days between the two given days in a year. Let's say one day is let us say second of February, the other day is let say 3rd of April. It says how many days are there between second February and 3rd April. The only condition is this the two dates 2nd February and 3rd April should be within this same year.

So, this program will obviously be manipulating data like a date right because it will ask you to enter 2 dates within the same year, and then it will calculate the number of days in between. Of course, it has to take smaller things like is the year a leap year. You know if it is a leap year then you know if February is included in the range of days to be calculated then it has to adjust the number of days and so on. So, this program has several preconditions. They are all listed to start with this comments here. So, it says day 1 and day 2 must be in the same year because that is what the program is meant to calculate if given across years this is not the program that will calculate.

The second pre condition says that the number that you enter for month it should be a number between 1 and 12. Both month 1 and month 2, the two months that told you right February and April, both these months should be numbers between 1 and 12; and it says both the days should be between 1 and 31 and it says in addition, month 1 should be less than or equal to month 2. In the sense that I do not ask for the number of days from let say 25th of May to 28th of February because its within the same year the first date that I

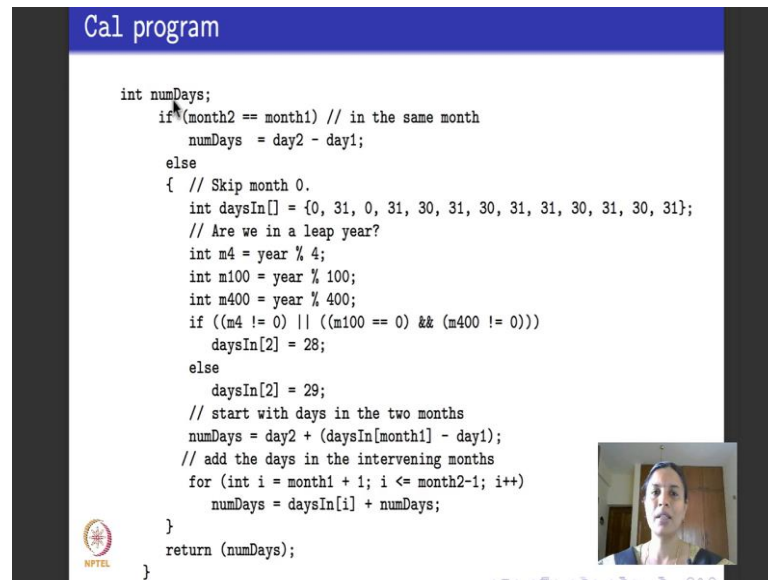
give should be before the second date that I give. So, a simple way of enforcing that is to say that I enter month 1 which is the part of date one and that should be less than or equal to month 2 which is a part of date 2. And then it says the range for year should be between one and ten thousand ,this is just another precondition.

Now, these are what are called preconditions. I hope it makes sense why they are needed because otherwise you could expect any kind of garbage as a input which we do not want. So, we write a whole set of pre conditions which says that please accept inputs only of these kinds. If the input is not of these kinds then you write appropriate exception handling statements which will make the program alert the user about the range or details being wrong. Now, one is to write these preconditions and the other is to check whether these preconditions themselves are complete. It is not very difficult to see that for this example that we have considered the pre conditions are not complete. In fact, they are reasonably inaccurate. Why is that so because here for example, it says for every day 1 as how is the date given date is given is day month and year and the condition for day 1 says that it is a number between 1 and 31.

So, let us say the month is a month of November we really do not have something like 31st November, where as these preconditions will let you enter something like 31st November, they do not specify that. Of course, in the program we will take care of the fact that does not happen, but these preconditions per se do not rule out that. Similarly, for a day in month like February, you can merrily go ahead and enter 29 February for a year that is not a leap year or for that matter you can enter 30th February and 31st February. These pre conditions let you do that. The other thing is the condition that we have asked is that both the days should be in the same year.

So, if this range for year is not really necessary data at all, but it is given just as a precautionary thing. On one side pre conditions are useful to rule out junk inputs, but on the other side preconditions can get very cumbersome if you have to get every detail right. So, pre conditions can be inaccurate, they need not be complete. So, this is how the Cal program starts.

(Refer Slide Time: 18:37)



So, it takes input as month 1, day 1, month 2, day 2 and year. So, it says in this year this is the range from day 1 and month 1 to day 2 and month 2, you calculate the number of days. How does the program work? Here is how the program works it call this thing method called Cal, it has an internal variable called number of days which is the variable that is returned back to the main program. What it first says is the if it is in the same month if the 2 days are in the same month that is if months 2 is equal to month 1 the number of days is easy to calculate. You just do day 2 minus day 1 then you get the number of days. Otherwise what it tries to do is to first tries to populate the calendar of how many days are there in a month.

So, this is the calendar that it populates. So, it says month, skip month 0 means skip this entry 0, then goes on as January, February, March, April, May, June, July and so on. So, January has 31 days. For February it is just initially put it a 0, based on a whether it is a leap year or not we will populate it as 28 or 29. March has 31 days, April 30 days, may 31 days, June 30 days, and so, on till December.

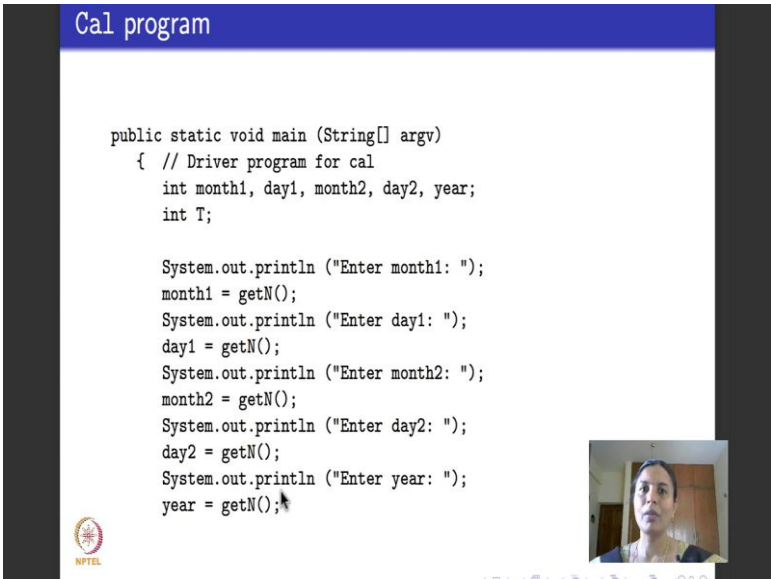
Now, the next piece that the code contains is to populate the date for February. So, it says, first checks whether are we in a leap year. So, year is already entered here as input, year is already entered here as input. So, it tries to divide a year by 4 by 100 and then by 400 and it computes the standard check for leap year right. If m does not, m 4 which is the year divided by 4 is not equal to 0 or m 100 which is the year divided by 100 equal to

0 and m 100 is not equal to 0 then it is not a leap year. So, the February which is this second; 0, 1, 2 the second the 3rd entry, but the second index entry in the list day days in is set to 28. If this condition is violated then it is indeed a leap year. So, you set the number of days in that month in the entry for February to be 29. So, after this you have populated this full array with the correct days for each month.

Now, what you do is you calculate the number of days in between the 2 days that was entered. First you calculate the days within the two months. Start with the days within the two months. So, you do number of days is day 2 plus days in month minus day 1, to this you add the days in the inter meaning intervening months right. So, let say I am asked to find out the number of days from second February to let say 25th May, I first start with the days in February and days in the month of may then add to that I add the number of days that I have in march and April that is what this segment of code is doing and for adding the number of days in the intervening month I start from the months after the month that I began and end in the month before the month that I ended and keep adding. Once you finally finish this you return number of days which tells you the number of days that exists in a month.

So, this is the total program for calculating the number of days in a month. This program had all these preconditions. These preconditions have to be brought into the program.

(Refer Slide Time: 21:41)



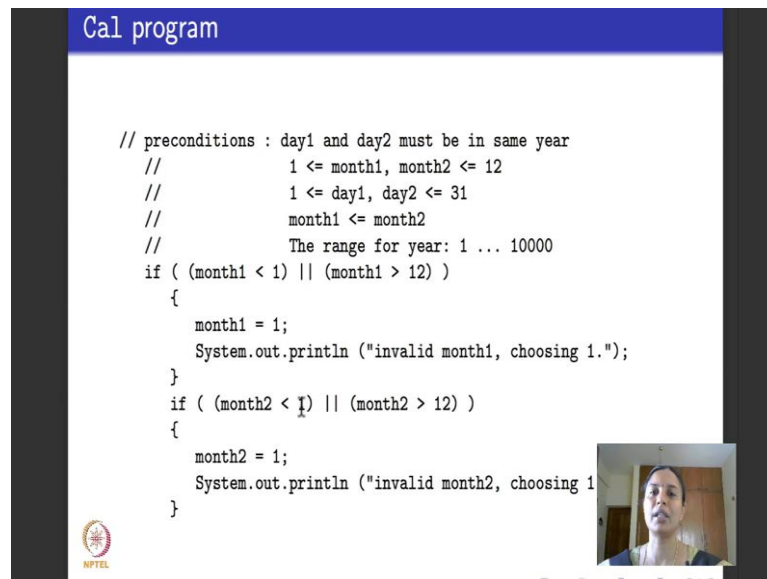
Cal program

```
public static void main (String[] argv)
{ // Driver program for cal
  int month1, day1, month2, day2, year;
  int T;

  System.out.println ("Enter month1: ");
  month1 = getN();
  System.out.println ("Enter day1: ");
  day1 = getN();
  System.out.println ("Enter month2: ");
  month2 = getN();
  System.out.println ("Enter day2: ");
  day2 = getN();
  System.out.println ("Enter year: ");
  year = getN();
```

So, we will continue writing the code. We will write a main program that takes as all the inputs and calls the program for Cal. So, this is a driver program that calls this method Cal. So, what does the main program do? It initializes these things, it initializes a variable called t, then it gets the data: enter month1, day1, month 2, day2 and year, it gets all this data.

(Refer Slide Time: 22:05)



```
// preconditions : day1 and day2 must be in same year
//               1 <= month1, month2 <= 12
//               1 <= day1, day2 <= 31
//               month1 <= month2
//               The range for year: 1 ... 10000
if ( (month1 < 1) || (month1 > 12) )
{
    month1 = 1;
    System.out.println ("invalid month1, choosing 1.");
}
if ( (month2 < 1) || (month2 > 12) )
{
    month2 = 1;
    System.out.println ("invalid month2, choosing 1.");
}
```



Now it has checks for these preconditions. I have just repeated the preconditions here just so that it is easy to refer to. What are the pre conditions ? They say that the day 1 and the day 2 must be in the same year, month 1 and month 2 should be numbers between 1 and 12 day, 1 and day 2 should be numbers between 1 and 31, month 1 should be less than or equal to month 2 and year should be a number between 1 and 10,000.

So, I write if statements for all these conditions. I say if month 1 is less than 1 or if month 1 is greater than 12 you throw an error. Similarly for month 2, if month 2 is less than 1 or if month 2 is greater than 12 you throw another error.

(Refer Slide Time: 22:43)

Cal program

```
if ( (day1 < 1) || (day1 > 31) )
{
    day1 = 1;
    System.out.println ("invalid day1, choosing 1.");
}
if ( (day2 < 1) || (day2 > 31) )
{
    day2 = 1;
    System.out.println ("invalid day2, choosing 1.");
}
while ( month1 > month2 )
{
    System.out.println ("month1 must be prior or equals to month2")
    System.out.println ("Enter month1: ");
    month1 = getN();
    System.out.println ("Enter month2: ");
    month2 = getN();
}
```



And go on with this, write conditions for day. If day 1 is less than 1 or day 1 is greater than 31 throw an error; if day 2 is less than 1 or day 2 is greater than 31 throw an error; if month 1 is greater than month 2 ask them to enter correct details, that is not acceptable.



(Refer Slide Time: 23:04)

Cal program

```
if ( (year < 1) || (year > 10000) )
{
    year = 1;
    System.out.println ("invalid year, choosing 1.");
}

T = cal (month1, day1, month2, day2, year);

System.out.println ("Result is: " + T);
}
```

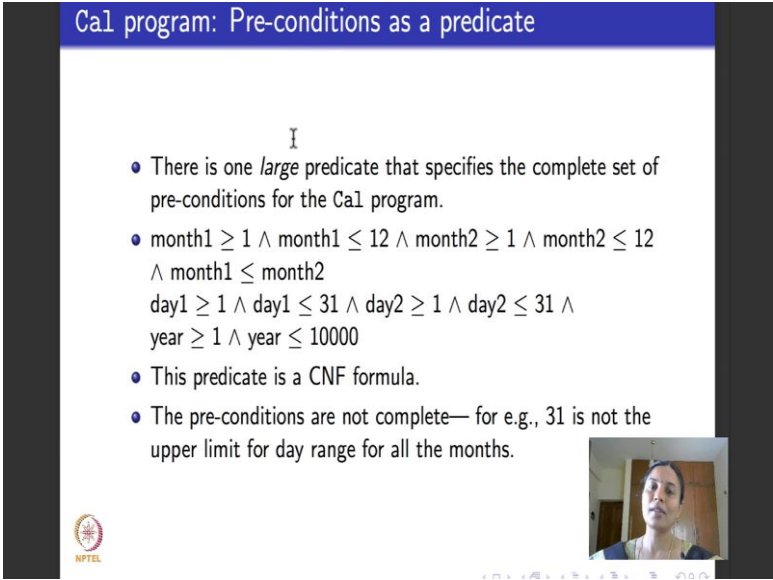


Like this you go on building program snippets that begin to check for each of the pre conditions. This checks for years, and finally, it calls the method calendar with all these inputs that have been validated and runs the method calendar. Whatever data calendar returns, it will print that data. So, this is how are typical program looks like and pre

conditions come as specifications that talk about what are the specifications or conditions related to inputs in the program. It is up to the programmer to be able to add these kind of conditions apart from the main functionality of the program to make sure that the inputs are well taken care of. Now if you go and see this program we have so many predicates here. There is this one, here this if statement, there is this if statement here in which are another predicate for month there are 2 predicates for days there is one more predicate for months there is one more predicate for year.

So, all these predicate mean that we can apply logic coverage criteria. So, what do we do, we collect them all.

(Refer Slide Time: 24:03)



Cal program: Pre-conditions as a predicate

I

- There is one *large* predicate that specifies the complete set of pre-conditions for the Cal program.
- $\text{month1} \geq 1 \wedge \text{month1} \leq 12 \wedge \text{month2} \geq 1 \wedge \text{month2} \leq 12 \wedge \text{month1} \leq \text{month2}$
 $\text{day1} \geq 1 \wedge \text{day1} \leq 31 \wedge \text{day2} \geq 1 \wedge \text{day2} \leq 31 \wedge$
 $\text{year} \geq 1 \wedge \text{year} \leq 10000$
- This predicate is a CNF formula.
- The pre-conditions are not complete— for e.g., 31 is not the upper limit for day range for all the months.

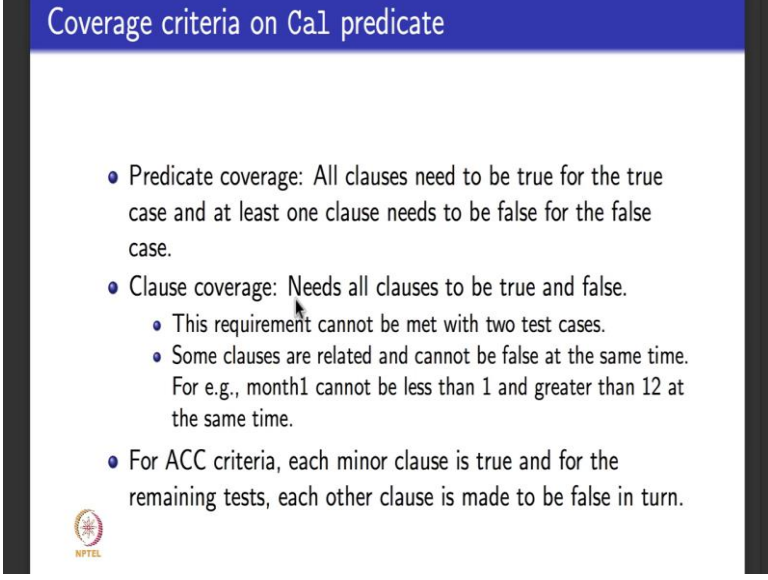
NPTEL

And we say these are all the constituent preconditions for this program. So, there is one large predicate that specifies the complete set of pre conditions for calendar program. That is what I have written out here as taken from the code. Tts again very easy to read. It says month 1 should be a number between 1 and 12, month 2 should be a number between 1 and 12, month 1 should be less than or equal to month 2, day 1 should be a number between 1 and 31, day 2 should be a number between 1 and 31, year should be within this range.

If you now look at these predicate, it is very easy to see that it s a formula in conjunctive normal form and this is what I told you earlier. These predicates has pre conditions still are not complete. That is a different problem we will worry about it later. For example,

31 is not the upper limit for the day of all months. This particular bit is taken care of in the code, but it need not be, the goal for us is not to check inaccuracy or incompleteness of predicates. The goal for us to check is that the predicative that is written does it make sense.

(Refer Slide Time: 25:08)



The slide is titled "Coverage criteria on Ca1 predicate" in a blue header. It contains three main bullet points. The first is "Predicate coverage: All clauses need to be true for the true case and at least one clause needs to be false for the false case." The second is "Clause coverage: Needs all clauses to be true and false," which has two sub-bullets: "This requirement cannot be met with two test cases." and "Some clauses are related and cannot be false at the same time. For e.g., month1 cannot be less than 1 and greater than 12 at the same time." The third is "For ACC criteria, each minor clause is true and for the remaining tests, each other clause is made to be false in turn." An NPTEL logo is in the bottom left corner.

- Predicate coverage: All clauses need to be true for the true case and at least one clause needs to be false for the false case.
- Clause coverage: Needs all clauses to be true and false.
 - This requirement cannot be met with two test cases.
 - Some clauses are related and cannot be false at the same time. For e.g., month1 cannot be less than 1 and greater than 12 at the same time.
- For ACC criteria, each minor clause is true and for the remaining tests, each other clause is made to be false in turn.

So, here is a predicate in conjunctive normal form. I told you how to generate test cases TRs for predicate in conjunctive normal form for the various active clause criteria. So, let us before we move on to active clause coverage criteria, let us look at a predicate coverage criteria. Predicate coverage criteria for such a predicate is quite easy: make all clauses true once, then the whole predicate will be true. Every clause has to be true because this is an And. Even if one clause is false the predicate will not be true and to make the predicate false you make any one arbitrary clause, at least one clause false the whole predicate will become false. You could make more than one clause false also.

So, predicate coverage is easy to do what about clause coverage? Clause coverage is a simple TR, as per definition says that each clause should be made true once and false once. But there is a catch here some clauses are dependent on each other. So, I cannot make each clause true once and false once. In fact, when we look at logical formulae corresponding to specifications will realize that while achieving clause coverage or while achieving active clause coverage, we will always have this condition this problem. In

several cases, there will be interdependency between the clauses. It could be the case that 2 clauses simultaneously cannot be true, 2 clauses simultaneously cannot be false.

So, given all these extra conditions on the clauses it could be very well the case that the test requirement can become infeasible or the fact that its infeasible could indicate an error in the specifications. Both are valuable information. Like for example, in this case suppose I want to achieve clause coverage, then some clauses are related as I told you and they cannot be false at the same time. For example, month 1 cannot be less than 1 and greater than 12 at the same time.

So, we have to be careful when we deal with clause coverage. In this case it does not mean that the pre precondition has an error, but in certain other cases it could mean to the precondition has an error. So, we when we want to do clause coverage based testing for such predicates, we always exercise a bit of caution because sometimes it may not be practically possible to get test cases for clause coverage. For ACC criteria I told you how to generate right. For conjunctive normal form you have a first row of all true then you have a diagonal of false values, as many as the number of clauses that you have. So that is the test requirement for ACC criteria very easy to get.

So that hopefully would have given you a good idea of how to work with specifications that come as preconditions and how to test for logical predicates based on that. Similarly, you can do for post conditions for invariants for assertions and wherever they are not available or given in English as a tester it is up to us to be able to derive write the logical specifications for them and generate test cases.

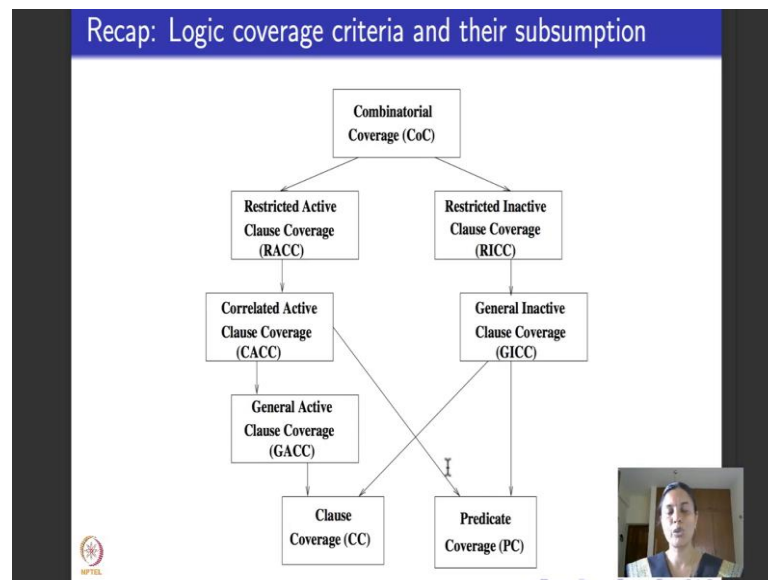
Thank you.

Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 29
Logic Coverage Criteria: Applied to finite state machines


Hello there. This is the last lecture of week 6. We will be done with logic coverage criteria with this lecture. Next week I will begin with the new test case design algorithm. What are we going to do today? We are going to look at how a logic coverage criterion applies to finite state machines.

(Refer Slide Time: 00:29)



So, this is same slide that I had shown you every lecture that we did in logic coverage criteria, meant to be a recap of the various coverage criteria that we saw and their subsumption. In today's example when we do finite state machines, we will see may two or three of these coverage criteria and see how to write test requirements in test cases for those, for specifications that come from finite state machines.

(Refer Slide Time: 00:55)



Finite state machines

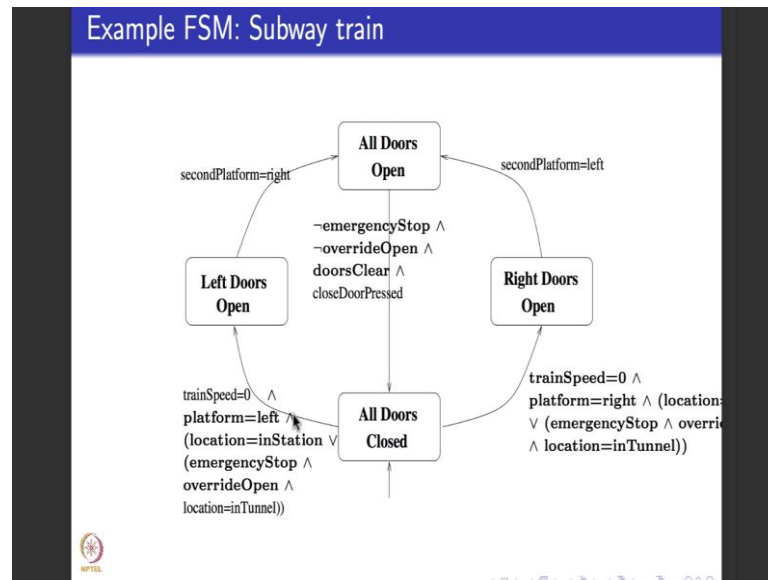
- Finite state machines as graphs occur as specification models, design models, in UML state diagrams etc.
- Transitions in finite state machines have guards or triggers, which are basically logical expressions/predicates.
- We will see how to apply logic coverage criteria over such FSMs through an example.

NPTEL

So, we introduce finite state machines when we did graph based testing. They have nodes or vertices which are states and the edges in finite state machines are called transitions. Edges are labelled with actions events and also guards, which are basically conditions that tell you that when an event can be happen, and when it happens the resulting action can happen.

Finite state machines have designated initial states. They may or may not have final states. They are very popular model for specification, for design, they occur as UML state diagrams. Especially in the embedded control software domain finite state machines are routinely used to specify control algorithms.

(Refer Slide Time: 01:39)



So, we will see how to apply logic coverage criteria for finite state machines using an example. So, here is an example of a very small finite state machine. In real life finite state machines that correspond to realistic models hardly are so small, but what people do was what is called modular design. They come up initially a very high level finite state machine that acts specification or is it design model. As and when they get clarity they keep refining or adding details of their finite state machines. So, even it is a small machine here that we are looking at in this example, it has enough details, enough predicates for us to be able to look at how logic coverage criteria applies to this.

So, this is a machine of, let us say, a model of how the doors of a subway train runs, subway train or a metro train. So, these trains keep moving from one station to the other, and the way they move, they have doors, the trains have doors on both sides. And typically for safety reasons, if the platform on a particular station comes on the left side then it is expected the only the doors on the left hand side open. The doors on the right hand side are closed. And if a platform, if particular station comes in such a way that there is platform accessible on both sides, then both sides doors open and when the train is moving, which means it is inside a subway tunnel, then you expect both doors to be closed.

So, here is a state machine of a subway train that has 4 states that talk about what are the status of all doors in the train. So, the initial state is the state which says all the doors of

the train are closed. From that state it could move to one of these two states. It could either move to a state where all the left doors are open or it could move to a state where all the right doors are open. And from either of these states, left doors open or all right doors open, it could move to a state which says all the doors are open. So, each of these moves or transitions are guarded by a whole set of conditions as you can see here.

For example, from the state all doors closed to the state left doors open, if I have to take that transition then all these conditions must be true. The final state machine has access to all these variables. Like in program you see the variables right in the beginning. Finite state machines they are not explicitly declared, but we assume that eventually this finite state machine design or specification is meant to be modeled, implemented using codes. So all these will surface as a real variables that will be in the code corresponding to the finite state machine.

So, there is a variable that talks about the speed of a train, called trains speed. It says that train speed should be 0, which means the train should not be moving. There is a variable which says where is the platform, is the platform on the left hand side or is the platform on the right hand side. So, in this case the guard to transition to left doors open, the platform should be on the left hand side and there is another variable which talks about the location of the station, location of the train. It is says the train is in station which means it is come inside the station, and it is either this or there is an emergency stop.

Somebody is pressed the emergency stop button and because this an emergency stop somebody is trying to open the door by overriding it, while the train is still in the tunnel. But the train is not moving in all the cases. So, there are so many variables and this large guard tells you when the transition from all doors close to left doors open happens. So, it is says train should not be moving, platform should be in the station and the train should, platform should be on the left side and train should be in station. Either this or there was an emergency stop, somebody is trying to override open the doors and the train is still in the tunnel.

So now, similarly from the state all the doors closed to the state right doors open, when do I go? Again the train should not be in moving, platform should be on the right, location should be in station, I am sorry this is gone out of the slide in little bit emergency stop, override open, location in tunnel.

It is an exact replica of this. The only condition that is changed is the platform is come on the right hand side for this transition. Now it is so happens that the second platform is on the right hand side and there is a platform on the right hand side then apart from the left doors you can open the right doors implicitly, and you go to a state which says all door are open. And similarly if there is a second platform, when the right doors are open, if there is a second platform on the left hand side then you could open the left doors also and go to a state all doors open. And from the state all doors are open to the state all doors closed, how do we go? We go when the doors are clear and somebody presses the close door button and there is no emergency stop and nobody has pressed override open.

So, is it clear how the simple state machine looks like and why I have had drawn at this way? W have drawn at this way because the focus of this lecture is to be able to illustrate the use of logical predicates as they come in finite state machines. So, this state machine even at this level could have actions. We have not really specified all that because I want to be able to focus on logical predicate. So, if you see there are so many predicates that we can consider, five of them here. Just for illustrate purposes let me take this predicate, the one on the bottom left which goes, which lets you go from the state all doors closed to the state left doors open.

We will take this predicate and illustrate the use of logic coverage criteria that we learnt on testing this predicate, indirectly testing the influence of that predicate or guard on the finite state machine.

(Refer Slide Time: 07:39)

Logical coverage criteria for Subway train FSM

- Consider one of the predicates, say,
 $\text{trainSpeed}=0 \wedge \text{platform}=\text{left} \wedge (\text{location} = \text{inStation} \vee (\text{emergencyStop} \wedge \text{overrideOpen} \wedge \text{location} = \text{inTunnel}))$
which occurs as the guard in the transition from the state of all doors being closed to the state of left side doors being open.
- This predicate has six clauses.
- Predicate coverage criterion on this predicate will dictate whether this transition can be taken or not and can be easily satisfied.
- We work out active clause coverage criteria for this predicate.

NPTEL

So, I have just written the predicate that I want to consider. So, this is the predicate, it says train speed 0, platform is on the left, the train is in station or somebody is pressed the emergency stop, doing override open while the train is still in tunnel. This occurs as a guard in the transition from the state of all doors being closed to the state of left doors being open. How many clauses does this predicate have? Let us count, this is one: train speed equal to 0, platform equal to left is a second clause, location in station is the third clause, emergency stop, fourth one, override open fifth clause, location is equal to in tunnel, sixth clause.

Predicate coverage criteria for this predicates is very easy. It will tell you basically whether you can take this transition or not. You can take this transition if this predicate is true which means if the clauses that are connected by an AND are true otherwise you cannot take the transition. So, that is what predicate coverage criteria will say. Now we will work out active clause coverage criteria for this predicate.

(Refer Slide Time: 08:54)

Conditions under which each clause determines the predicate

- For clause $\text{trainSpeed}=0$: $\text{platform}=\text{left} \wedge (\text{location} = \text{inStation} \vee (\text{emergencyStop} \wedge \text{overrideOpen} \wedge \text{location} = \text{inTunnel}))$
- For clause overrideOpen : $\text{trainSpeed} = 0 \wedge \text{platform} = \text{left} \wedge (\neg \text{location} = \text{inStation} \wedge \text{emergencyStop} \wedge \text{location} = \text{inTunnel})$.
- We can similarly compute for all the other clauses.

Page 6 of 11

So, I take this predicate, it has 6 clauses. What is the first step to do to determine active clause coverage criteria? Per clause I have to say call it as a major clause and see when that clause determines the predicate. So, if it is one of the clauses in the predicate, I do what is called determining p subscript a . So, I have worked it out offline. I seriously urge you to take this predicate and work out for each of the 6 clauses when that clause will determine the predicate using the formula that says replace the clause with true in the predicate, replace the clause which is false in the predicate and XOR them. That is the formula if you remember.

So, I have used that formula I have worked it out and for the first clause in the predicate, which is train speed is equal to 0 after simplification, this is the p_a value, that is this is the p train speed is equal to 0 value. And similarly I have also worked out for the clause override open after doing the same formula, substituting override open with true one, substituting override open with false once, and XOR-ing them and simplifying the resulting formula, I will get this simplified predicate.



Similarly, for all the remaining four clauses we can compute if that clause is the major clause, the conditions under which that clause will determine the predicate. I have just done two of them here, the remaining four can be worked out. In fact, for these two also I have given you the final resulting formula, I have not given you the steps of working.

(Refer Slide Time: 10:23)

TR: CACC for the predicate

Major clause	Speed=0	platform =left	inStation	emergency Stop	override Open	in Tunnel
trainSpeed=0	T	t	t	t	t	t
trainSpeed!=0	F	t	t	t	t	t
overrideOpen	t	t	f	t	T	t
¬overrideOpen	t	t	f	t	F	t

We can compute CACC TR for other predicates in a similar fashion.



So, we take the first one which is train speed 0 and the other one that we worked out which is override open, and remember what are these? This is like a formula in conjunctive/ disjunctive normal form that we saw in the last lecture. So, it is very easy to write ACC test requirement for these formulas.

What you do? You make all the other clauses to take value true and the clause that is your major clause, make it true once make it false once. Similarly when our override open is a major clause you make it true once, you make it false once. The rest the clauses are all may true provided the diagonal, along the diagonal, it is so happens that override open is the fourth clause in this predicate. One, two, three, four, fifth clause, sorry it is the fifth clause in this predicate. So, the diagonal corresponding to that will be false the rest of the values will all be true.

So, this is how you write correlated active clause coverage criteria TR for that. I have just given you for the two clauses for which we worked out here and give when that clause determines the predicates. There are four more clauses, after you work them out you can fill up this table and see how it looks like. So this the TR for CACC for the predicate. So, to be able to give test cases for this TR, you have to make train speed value 0, if it ensure that the platform is said to left, in station as a Boolean variable should be made true, emergency stop should be made true, override open should be made true, and in tunnel should be made true. This how a test case look for that will look like.


(Refer Slide Time: 11:51)

One problematic predicate in the FSM

- While writing CACC TR for the predicate inStation, we encounter a unique problem with making the TR feasible.

Major clause	Speed=0	platform =left	inStation	emergency Stop	override Open	in Tunnel
inStation	t	t	T	f	f	f
¬inStation	t	t	F	f	f	f

- The FSM model has only two locations: inStation and inTunnel. They both are false in one TR (second row above), this is not possible.
- This could be an error in the model.



So, now you will realize there is a problem here. If you think a little bit about it, in this example itself we do have a bit of a problem. If you see for train speed 0 to be the major clause and for it to be influencing the predicate, look at the test case that. Each of these values should be made true. Now if you see is it actually feasible to make each of them true in the realistic scenario that the finite state machine would be implemented in, it will not. Why is that so, because if you see it says in station is set to true and in tunnel is also said to true in this both these cases; which means that the train is in the station and in the tunnel at the same time. That is not practically possible right, it is either in the station or in the tunnel. So, there could be a possible error or you could interpret it as that the finite state machines that we saw at its level of abstraction, does not give enough details to be able to specify what we need.

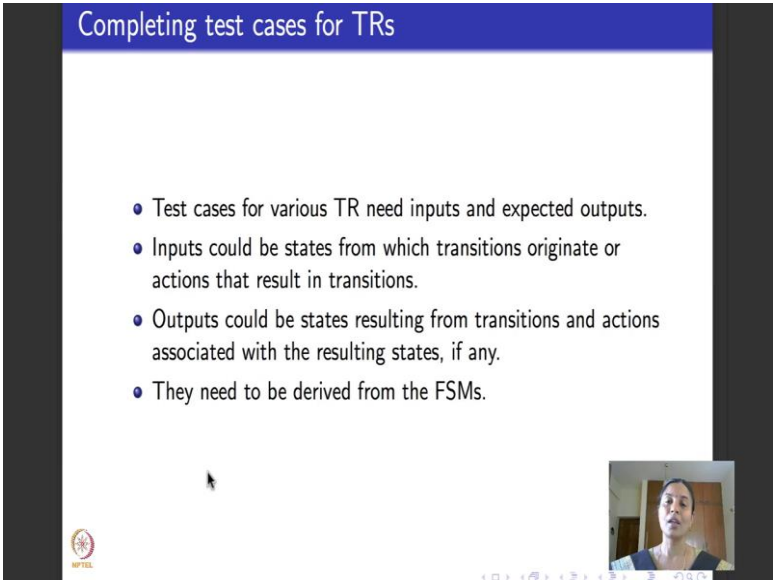
In fact, you could find several other errors. Here is one more. Suppose I try to write CACC TR test requirement for another clause, in station, which is here, which is one of the clauses that come in the predicate. So, here again you will realize that this is the major clause, I make it true once false once, the rest of it take true false values. If you see what I have done here is it says in station is false here and in station is a false here which means it says the reverse what the previous thing said. It is saying that the train neither in station nor in tunnel it is just disappeared. That also is wrong right there is something incomplete in the specification. This one said the train is both in the station and in the

tunnel, not clear, this says that the train is neither not in the station not and not in the tunnel. In the second row here if you see both are marked false that is also not possible.

So, it could be interpreted as two things, you could say that in this finite state machine model I have found two errors. There is inconsistency in specifying exactly where will the train be the predicates or guards do not really factor in the consideration that the train can be in either in the station or in tunnel. So, these guards are not clearly specified and that there is an error in the finite state machine.

Or another way to be treated it as is to say that there is incomplete, this is incomplete because the way it is specifies it does not give details about train being in exactly one of the location. So, more details have to be added or that the finite state machine has to be refined it to be able to complete that. Either way whatever it is we have found a possible bug in model and logic coverage criteria are very useful to be able to do this.

(Refer Slide Time: 14:47)



Completing test cases for TRs

- Test cases for various TR need inputs and expected outputs.
- Inputs could be states from which transitions originate or actions that result in transitions.
- Outputs could be states resulting from transitions and actions associated with the resulting states, if any.
- They need to be derived from the FSMs.

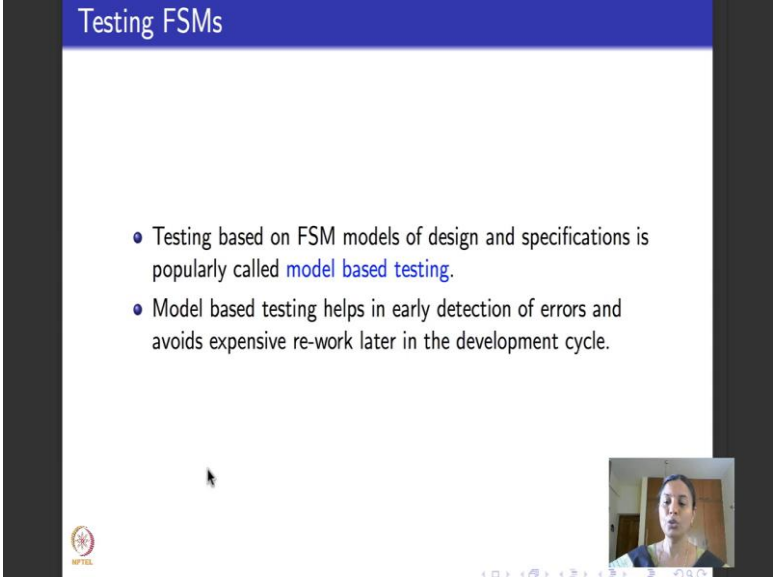
The slide is a presentation slide with a blue header. It contains a bulleted list of four points. In the bottom right corner, there is a small video inset showing a person speaking. At the bottom left, there is a small logo for NPTEL.

So, similarly if you work out for the other remaining clauses, you could find more inconsistencies and more errors. I just did two because in this particular case just these two were good enough to find inconsistencies or potential errors in the finite state machine as a specification. But there are, could be a couple of other issues also. Now when we are looking at an code, like when we saw logic coverage criteria as it applies to code, when we saw the two example the thermostat and the triangle type, we told you

that if a particular logic coverage criteria has internal variables, how to solve it for them how to ensure reachability.

But when we apply logic coverage criteria for finite state machines, as I told you when I introduced to you this example there is no explicit notion of inputs and outputs. So, what is a test case? Test case is directly given in terms of inputs and outputs that the finite state machine looks like, and output could also be given in terms of which is the resulting state. Sometimes they could be things wrong in the state that the Turing machine goes into even though state is not an explicit variable that occurs as a guard. So, it is up to the tester to infer all these information from informal specifications of finite state machines or semi formal specification of finite state machines, and complete the test case to identify a potential error or fault in the specification. They have to be explicitly derived, you cannot expect them to be present and readily available as they are with code.

(Refer Slide Time: 16:15)



Testing FSMs

- Testing based on FSM models of design and specifications is popularly called **model based testing**.
- Model based testing helps in early detection of errors and avoids expensive re-work later in the development cycle.

So, usually testing based on finite state machine is a very popular testing, there are huge number of papers that are in this area. Broadly called as model based testing, it helps in early detection of errors. Why it so, because finite state machines typically a models of specification. Specification and design are done before we write code. So, I have found the potential error writes there before I write code instead of finding it much later after I write code. It helps in early detection of errors if I detect it early rather than detecting it late, I save myself a lot of trouble with rework. So, it is saves lot of cost and lot of time

and you can check out the testing literature for a huge amount of material related to test case design and finite state machines, we have just covered logic base testing.

So, this brings us to an end of logic base testing. There will be an assignment this week that will deal with logic base testing for code, specifications and finite state machines. I will try and upload video next week that will let you solve tell you how to solve that assignment, but feel free to do it before you see that video.

Thank you.

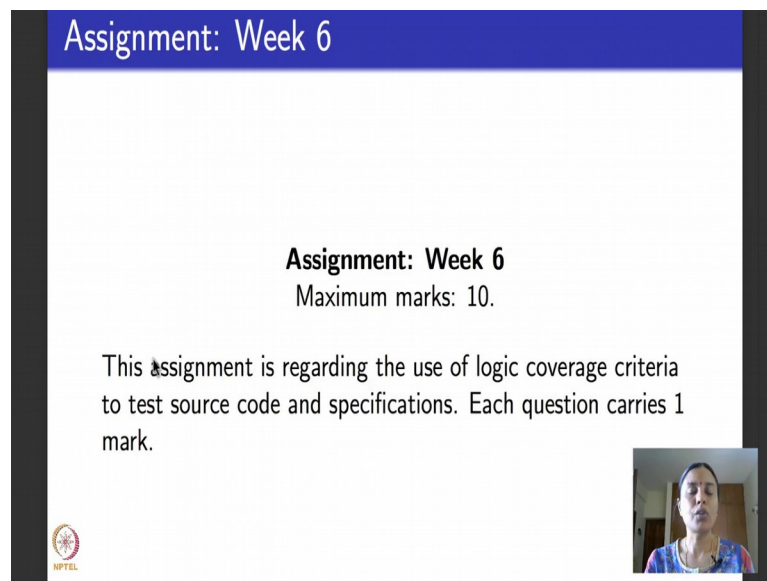
Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 30
Assignment Solving

Hello all welcome to week 7. The first thing that we will do this week, I will help you to go through the assignment that was uploaded for week 6, and then help you to solve that assignment.

If you remember what that assignment was, week 6 we finished logic coverage criteria. Specifically, we saw how logic coverage criteria applies to test source code to test design aspects; things like preconditions or post conditions or invariants that come with design. And we also saw an example of how logical predicate come as guards in finite state machines and saw how to design test cases based on that.

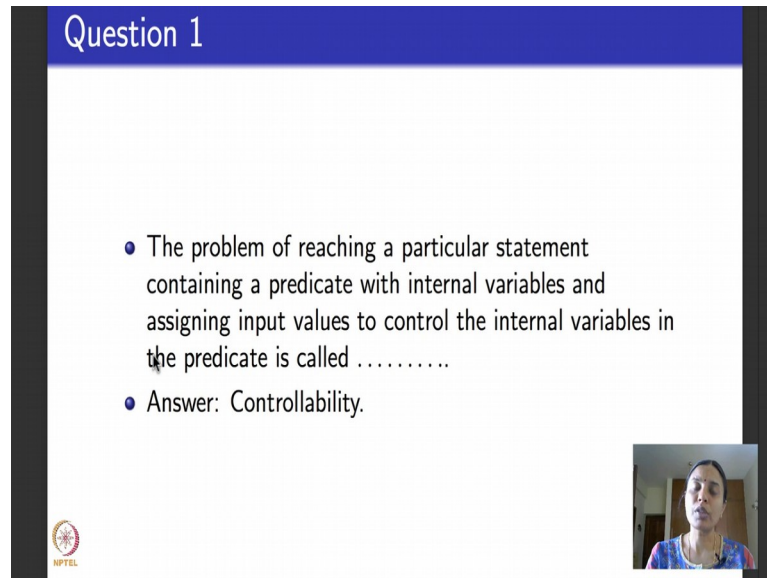
(Refer Slide Time: 00:53)



The slide has a blue header with the text "Assignment: Week 6". The main content area is white and contains the following text: "Assignment: Week 6", "Maximum marks: 10.", and "This assignment is regarding the use of logic coverage criteria to test source code and specifications. Each question carries 1 mark." In the bottom right corner, there is a small video inset showing a woman speaking. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.

So, given you a simple assignment for 10 marks consisting of fill in the blanks and objective choice questions for that week. This video will walk you through the assignment. And we will discuss how we could solve it. I hope all of you have made an attempt to solve the assignment yourself before viewing this video.

(Refer Slide Time: 01:10)



Question 1

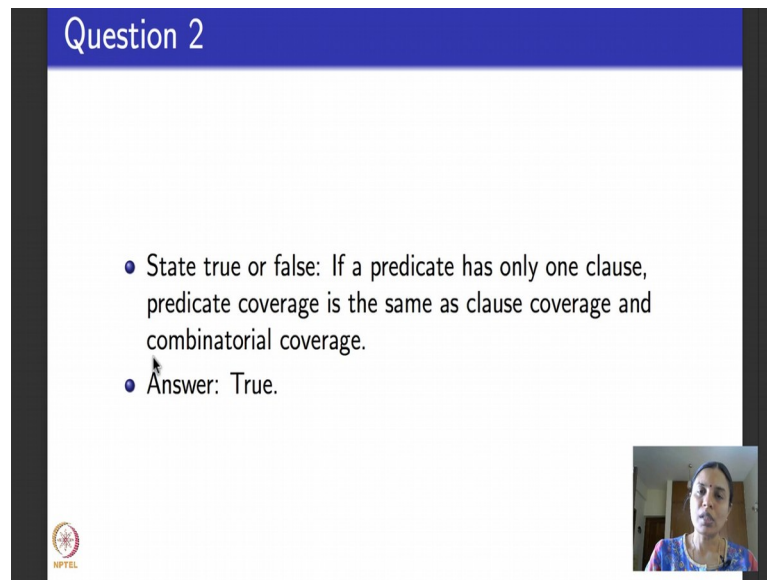
- The problem of reaching a particular statement containing a predicate with internal variables and assigning input values to control the internal variables in the predicate is called
- Answer: Controllability.

NPTEL

So, what was the first question in the assignment? It was a problem of reachability and controllability or R I P R criteria, as we called it in the first week. So, some predicates in the source code purely contain internal variables, may not contain any inputs. So, I need to be able to reach those predicates and effect whatever coverage criteria that I want to effect on them by changing values of inputs.

So, this question is about that, it asks you the problem of reaching a particular statement containing a predicate with internal variables, and assigning values to input such that the predicate can be made true or false, a particular clause in the predicate can be made true or false, what is this problem called. It is the problem of controllability. It comes after the problem reachability. Reachability will just reach the predicate. Controllability will help you to assign values to input variable so as to control the predicate.

(Refer Slide Time: 02:10)



Question 2

- State true or false: If a predicate has only one clause, predicate coverage is the same as clause coverage and combinatorial coverage.
- Answer: True.

NPTEL


The second question was a simple state true or false question, that asks consider a predicate that has only one clause. It asks you to say is clause coverage the same as predicate coverage is the same as combinatorial coverage? The answer to this question is an obvious yes or true, why because the predicate has only one clause. So, making the predicate true or false will also make the clause true or false, and there are exactly two possible assignments for this single predicate that contains a single clause, that is the entire truth table. The single predicate is made true once single predicate is made false once.

So, the truth table also has exactly only two values. So, it is the same as all combinations coverage. So, whenever a predicate has a single clause, all these three coverage criteria--- predicate coverage, clause coverage and all combinations coverage turn out to be the same.

(Refer Slide Time: 03:03)

Question 3

- Which of the following represents the normal form in which predicates corresponding to pre-conditions occur?
 - 1 Prenex normal form
 - 2 Disjunctive normal form
 - 3 Conjunctive normal form
 - 4 None of the above
- Answer: Third option above.



Question number 3 is a multiple choice question, it reads as follows. It asks you the following: which of the following represents the normal form in which predicates corresponding to pre conditions occur. There were four options given to you in the question, the option 1 was prenex normal form, option 2 was a disjunctive normal form, option 3 was conjunctive normal form, an option four says it occurs in a normal form that is none of the above three.

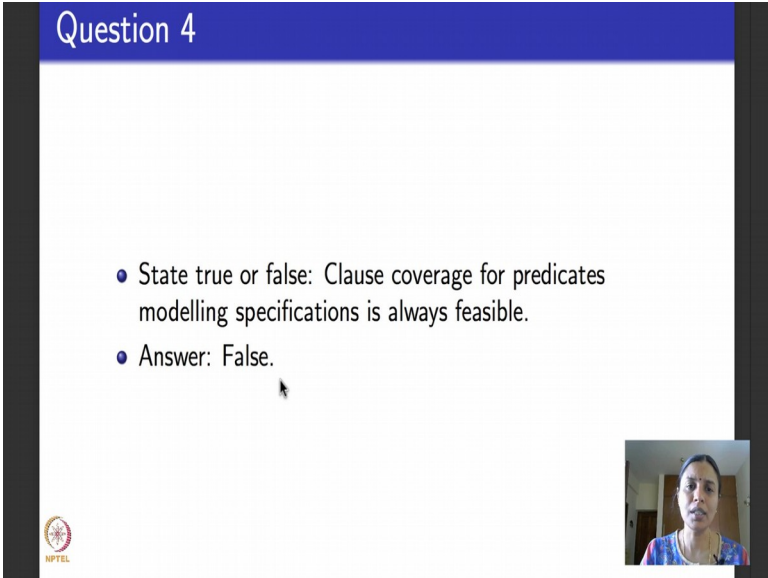
The correct answer to this is the third option which is conjunctive normal form. Why is that so? If you remember when we had seen predicate conditions, we have seen an example relating to these, the example program that we had seen was calculate the number of days between two given dates in within the same year and then the program has several pre-conditions which specify what the values for months will be like, what the values for the various dates will be like. And the fact that the value for the first month should be before the value for the second month, so that I can specify the number of days that have passed in between the first date and second date.

We had taken all these pre conditions and in the program that is the pre conditions that were written in English originally were transformed into predicates. Each condition was written as a simple clause and the entire precondition corresponding to the program was an “and” of all these individual clauses. So, if it is an “and” of all the individual clauses or if it is an “and” of what is called disjunct where inside there are “or”s, than the normal

form is supposed to be conjunctive normal form that is condemned represents and this entire predicate which occurs is a precondition is an “and” of all the individual conditions because we really cannot write an “or”, because you cannot say that one of the pre conditions should be true. It is just that each and every single precondition should be true.

So, the natural connective is an “and” when the connective is an “and”, the normal form that we are talking about is conjunctive normal form. Disjunctive normal form deals with “or”s. Prenex normal form, inn fact, does not deal with this kind of thing, it deals with formulas which have quantifiers, I have just given it as a filter option. So, the correct option here is conjunctive normal form.

(Refer Slide Time: 05:13)



Question 4

- State true or false: Clause coverage for predicates modelling specifications is always feasible.
- Answer: False.

NPTEL

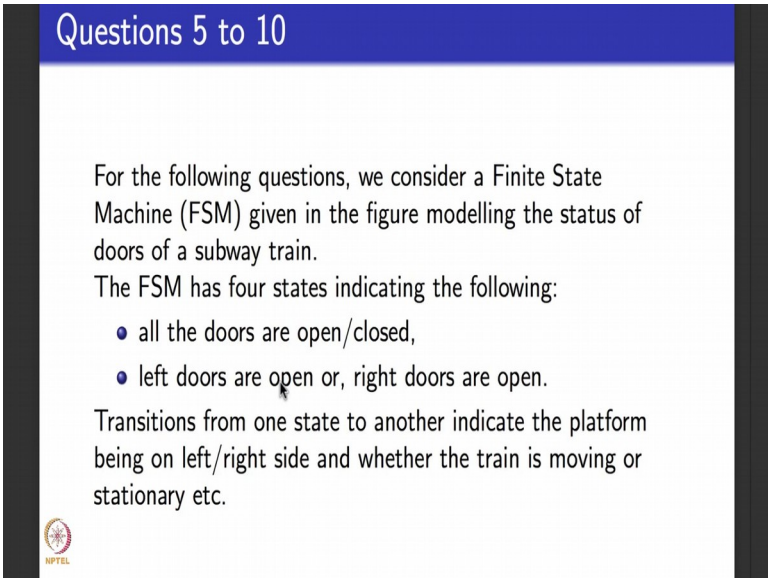
So, the next question is asked you to state at the whether of given statement is true or false. Te statement that was given is this--- clause coverage for predicates, that model specifications always visible? In other words when we consider specifications which are requirements given in English or as finite state machines, write out the logical formula corresponding to them and consider clause coverage for these predicates. Is it always possible to write test cases for the TR of clause coverage or in other words is clause coverage feasible?

The answer is false. Why is it false? We saw one example through the course of lectures last week. If you remember we saw the example of a finite state machine corresponding

to a subway train, and there, using illustrations I showed you how if one clause becomes true then the other one cannot be true.

So, two clauses in a particular predicate cannot be true simultaneously. So, if I have to consider clause coverage, I have to be able to make each clause in turn true and false, but they occur in such a way that if one clause is true the other will never be true. So, if I try to achieve clause coverage for one clause, I will not be able to achieve clause coverage for the other clause. So clause coverage for predicates that occur as specifications are not always feasible.

(Refer Slide Time: 06:38)




Questions 5 to 10

For the following questions, we consider a Finite State Machine (FSM) given in the figure modelling the status of doors of a subway train.

The FSM has four states indicating the following:

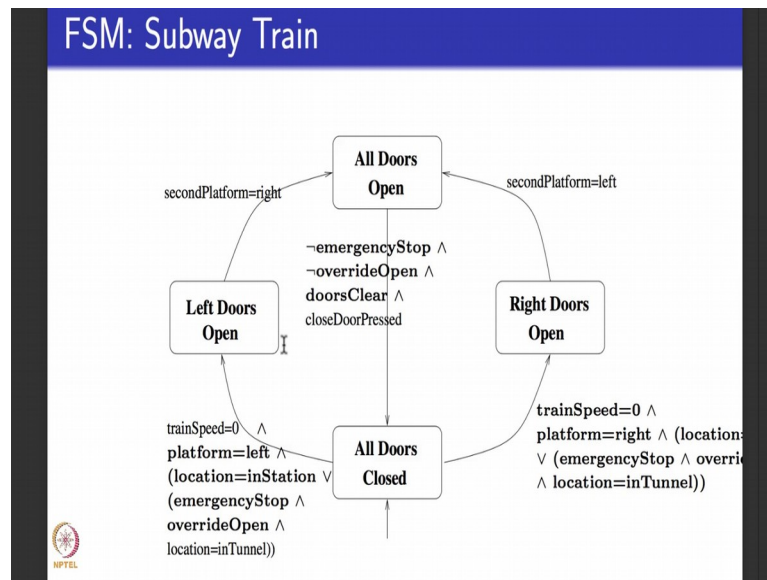
- all the doors are open/closed,
- left doors are open or, right doors are open.

Transitions from one state to another indicate the platform being on left/right side and whether the train is moving or stationary etc.



So, the answer to this question is false. For the remaining 6 questions, that is question number 5 to question number 10, I had given the state machine that we saw during the lectures which was the state machine corresponding to a subway train and had asked you to work out a few things related to predicate coverage for that state machine.

(Refer Slide Time: 06:59)



So, the state machine has 4 states just to recap, these are the 4 states — all doors closed, all doors are open, the doors on the left hand side are open and the doors on the right hand side are open. Initially in that train the initial state as marked like this, is the state where all doors are closed. And the train starts moving out of the station when it reaches the next station, if the platform is on the left hand side then the left doors becomes open, when the platform is on the right hand side the right doors become open. In case there is platform that is accessible on both the sides then all the doors become open.

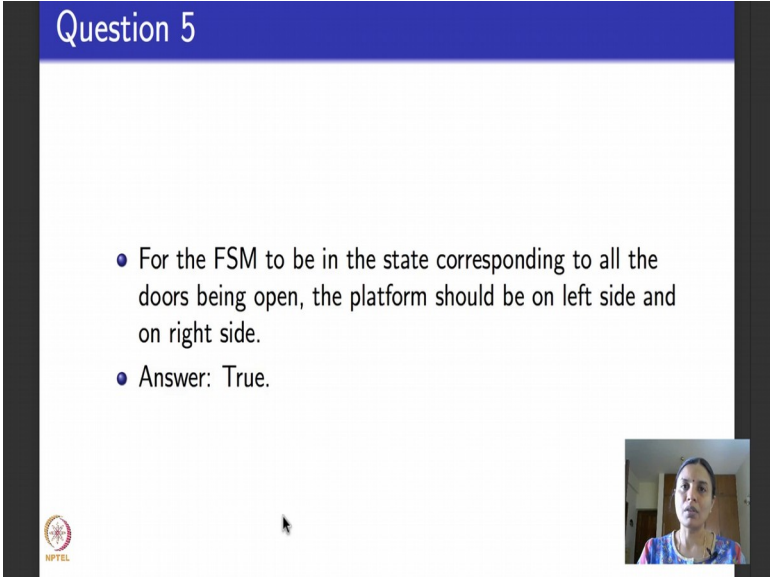
So, what are these predicates that model the transitions between the trains, between the states? So, it is say from the state all doors closed to the states left door open, I can do the transaction provided this fairly big predicate is true. So, let us read the predicate and understand what it says.

So, it says train speed is 0 which means the train is not moving, it is come to a station. If train is moving you better not open the doors right and it says the platform is on the left hand side that is what I told you a little while ago. The location is either the state the train is in the station which says location is in station or somebody has pressed an emergency stop button that is why the train is not moving, and has also done an override open of the train, while the train is still in tunnel. Is it clear please? Similarly on the other side I am sorry this thing, guard is running over a bit out to the slide, but it is an exact flip of the guard on the left hand side.

So, you see from a state where all doors are closed, I can move to a state where the right door is open provided the train is not moving, platform is on the right and the train is in the station or, somebody has pressed an emergency stop, done an over ride open for the door and the train is still in the tunnel. If any from any of these states the other door is open, the platform is available on the other side like if the right doors are open and the second platform is available on the left hand side then you can open the left door so, you got a state all doors.

Similarly if the left doors are open, and the second platform is available on the right hand side then you can open the right doors also and then you go to the state all doors open. So, from the state all doors open I can go to all doors closed, provided there is no emergency stop, nobody has pressed override open, all the doors have been cleared in the sense that there are no passengers moving in and out the doors and somebody has pressed the close door button. So, is it clear please what the state machine looks like? This is a simple four state machine studded with a lot of predicates. So, we will see some questions about the state machine and about predicate logic coverage criteria.

(Refer Slide Time: 09:52)



Question 5

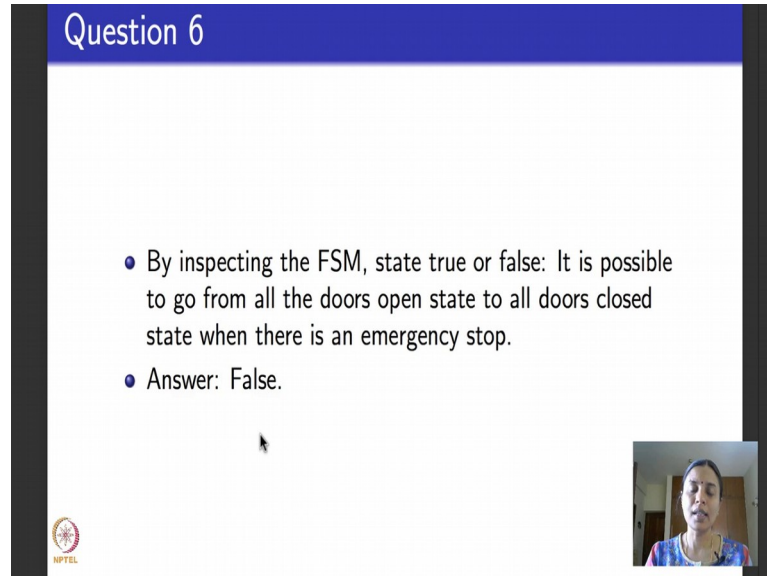
- For the FSM to be in the state corresponding to all the doors being open, the platform should be on left side and on right side.
- Answer: True.

NPTEL

So, the fifth question which was the first in the list of questions about the state machine, reads as follows. So, it is for the finite state machine to be in a state corresponding to all the doors being open, the platform should be on the left hand side and on the right hand side. Is your answer true or false? The answer is obviously, true because if you inspect

this to finite state machine as we saw, it could be in left doors open or right doors and if the platform is available on the other side then it goes to all doors open. So, the answer to this is true.

(Refer Slide Time: 10:26)



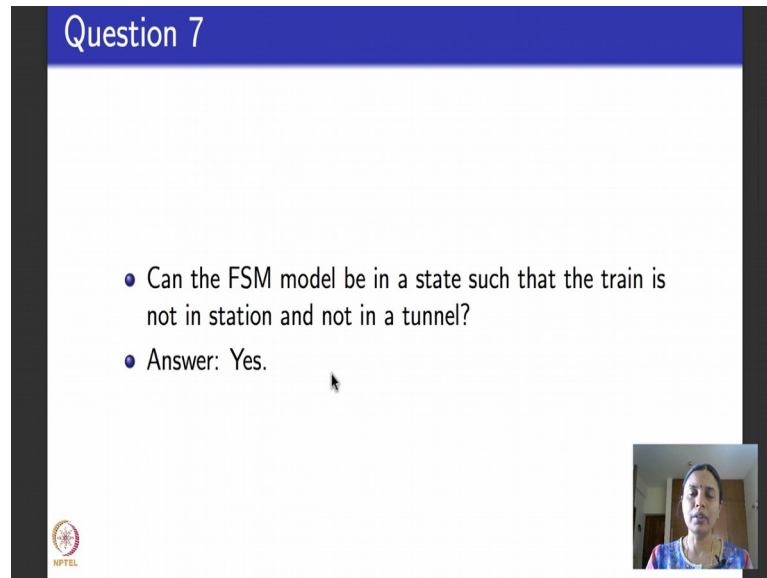
Question 6

- By inspecting the FSM, state true or false: It is possible to go from all the doors open state to all doors closed state when there is an emergency stop.
- Answer: False.

So, the next question says, look manually inspect the FSM which means study the FSM and state true or false for the following question. It is possible to go from all doors open state to all doors closed state when there is an emergency stop. So, let us go back and inspect the Turing machine, the state that we are considering the transition from out of is the state all doors open, the state that we are considering to translation into is the state all doors closed; and the questions says I can go from this state to this state when there is an emergency stop.

So, let us look at the guard in that transition there is an emergency stop clause in the guard in the transition, but what is the clause come with? It comes with a negation. If you look at it here the first clause is the clause on emergency stop it is a negation of emergency stop, not emergency stop. So, the answer to this question is false.

(Refer Slide Time: 11:23)



Question 7

- Can the FSM model be in a state such that the train is not in station and not in a tunnel?
- Answer: Yes.

NPTEL

So, moving on the next question question number 7 in the assignment was the following. Can the finite state machine model be in a state such that the train is not in station and not in a tunnel? Is that possible? If you remember we had looked at it even when we did the lectures, it is possible. So, when you try to do the truth table for each of the clauses you will realize that there is a particular state, where the not in tunnel, in tunnel need not be true and in station need not be true.

Whereas, in real life that is not possible and in fact when we did the lectures we had understood that there is something wrong with finite state machine. So, from the point view of testing it should be flagged as a potential error or vulnerability back to the designer.

(Refer Slide Time: 12:10)

Question 8

- Consider the guard:
 $\text{trainSpeed}=0 \wedge \text{platform}=\text{left} \wedge (\text{location} = \text{inStation} \vee (\text{emergencyStop} \wedge \text{overrideOpen} \wedge \text{location} = \text{inTunnel}))$.
This predicate has six clauses. Answer the following questions with respect to this predicate.
- Determine the conditions under which the clause $\text{trainSpeed}=0$ determines the guard.
- Answer: $\text{platform}=\text{left} \wedge (\text{location} = \text{inStation} \vee (\text{emergencyStop} \wedge \text{overrideOpen} \wedge \text{location} = \text{inTunnel}))$

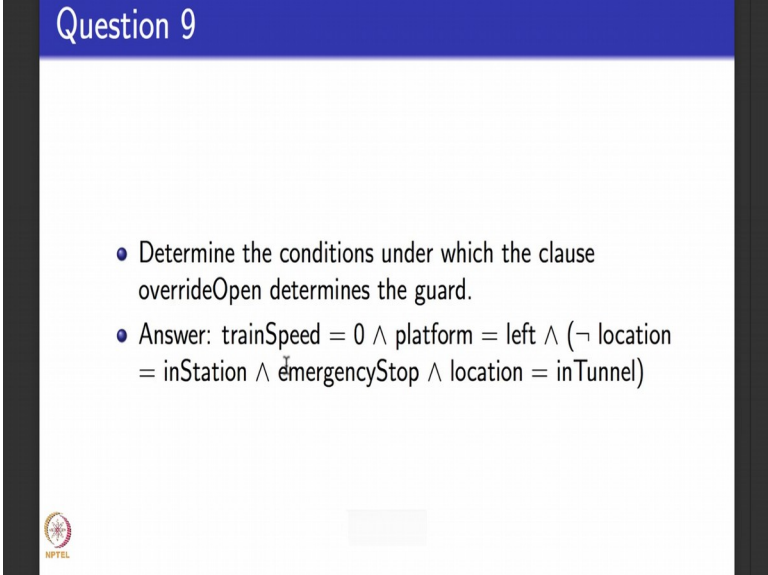
So, the next question was about guard clauses in a predicate determining the predicate. So, the we were asked to be started with one guard which is the guard that takes you from all doors closed to left doors open, which is this guard it is says train speed 0 platform is on the left hand side train is in station or there is an emergency stop, an override button pressed and the train is still in tunnel. If you see there are 6 guards here, 6 clauses sorry here 1, 2, 3, 4, 5, 6. What it asks you to do it takes we have taken examples of 3 clauses and ask you to determine the condition about when the clause will determine the predicate.

So, in this question we have taken the first clause, which is the clause of train speed being equal to 0, and you are asked to determine when this clause will determine the predicate. I have not given you the working out here, but the kind of thing that you have to do is substitute for the this clause to be true in this whole predicate, substitute for this clause to be false in the whole predicate and XOR it. Then use the logical operators that we know inference rule that we know to be able to simplify it. So, when I substitute it with true it will be true “and” something else.

So, the true will go away this one will get retained, when I substitute it false it will be false “and” something else. So, the false will stay back and then you go on simplifying this will be the final answer. It will be platform is equal to left and location in station “or” emergency stop and override open and location in tunnel. This is what will happen

this is the predicate that will get if you try to make the first clause determine the guard. So, you stop at this you, do not have to write test requirements and test cases for CACC and other active clause criteria, it just asks you to tell the conditions or the predicate under which this clause will determine the predicate that is all is the question about.

(Refer Slide Time: 14:08)



Question 9

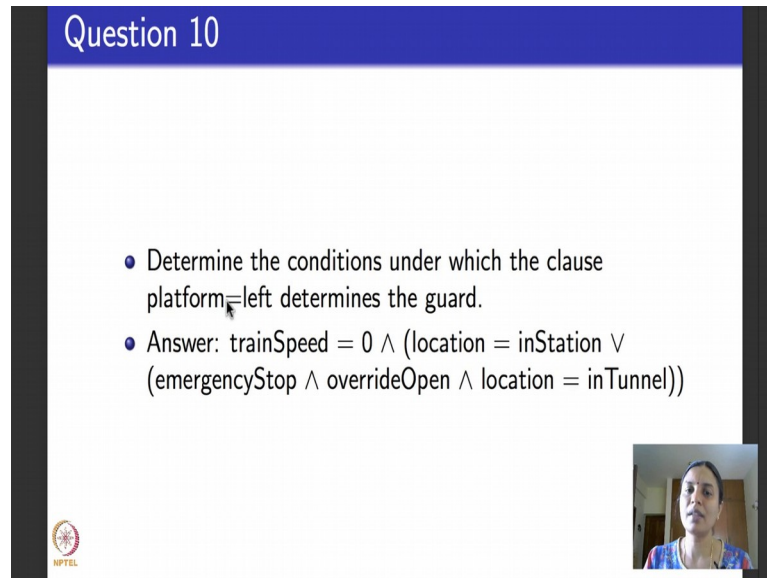
- Determine the conditions under which the clause `overrideOpen` determines the guard.
- Answer: $\text{trainSpeed} = 0 \wedge \text{platform} = \text{left} \wedge (\neg \text{location} = \text{inStation} \wedge \text{emergencyStop} \wedge \text{location} = \text{inTunnel})$

NPTEL

Next question is very similar, instead of taking, it takes the same predicate and it takes this clause `override open` which comes inside this hand and ask you to determine the conditions under which this clause will determine the predicate. So, you have to do the same thing substitute `override open` with `true` once, substitute `override open` with `false` once and then simplify the resulting logical formulae using the inference rules of logic and write the final predicate which will tell you exactly when this clause will determine the predicate.

Here again you do not write the test cases because there is no TR given in terms of do this to achieve certain active clause coverage criteria or inactive clause coverage criteria. So, you just stop at writing the resulting predicate and leave it.

(Refer Slide Time: 14:55)



Question 10

- Determine the conditions under which the clause $\text{platform} = \text{left}$ determines the guard.
- Answer: $\text{trainSpeed} = 0 \wedge (\text{location} = \text{inStation} \vee (\text{emergencyStop} \wedge \text{overrideOpen} \wedge \text{location} = \text{inTunnel}))$

Question number 10 is again a same thing. It picks up another clause this time the clause that is picked up is the clause which says platform is on the left, and it asks you to determine the conditions or the predicate under which the clause determines the guard . Do the same thing substitute this to be true once, substitute this to be false once XOR, then use the inference rules of logic to be able to simplify it this is the result in answer that you will get. Try and do it on your own if you get stuck feel free to query me in the forum and I will try to work it for you through a reply.

I hope this small assignment solving exercise was useful for you and your answers did match a lot with the answers that we have worked out. What you will do today, for rest of this week is I will start with a new chapter which is called input space partitioning. So, my next lecture, we will not do logic coverage criteria anymore. We will begin with the new set of test case algorithms based on input space partitioning.

Thank you.

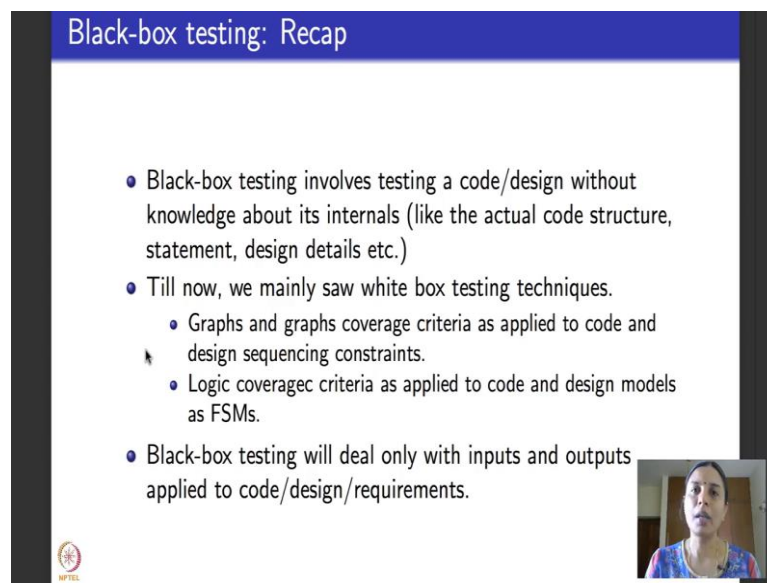
Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 31
Functional Testing

Hello all. This is week 7, second lecture. Last time we saw how to solve assignment 6. Now I am going to begin a completely new module in this course.


So far what did we see? We saw testing based on graphs and how it applies for source code, for design, for requirements. Then we saw testing based on logical predicates, how it applies to source code, for specifications and for finite state machines. Now we are moving into what is called, popularly called, functional testing or black box testing. And we will see this technique called input space partitioning, coverage criteria based on input space partitioning and look at an example of how to use these things.

(Refer Slide Time: 01:01)



Black-box testing: Recap

- Black-box testing involves testing a code/design without knowledge about its internals (like the actual code structure, statement, design details etc.)
- Till now, we mainly saw white box testing techniques.
 - Graphs and graphs coverage criteria as applied to code and design sequencing constraints.
 - Logic coverage criteria as applied to code and design models as FSMs.
- Black-box testing will deal only with inputs and outputs applied to code/design/requirements.



So, when we talking about functional testing, we are basically within the domain of black box testing. So, in the first week if you remember I had told you that there is this popular categorization in testing: white box and black box. So far the kind of things that we looked at graphs and logic. Whenever we apply it to code we were doing white box testing because we were talking about coverage criteria which exercised certain parts of

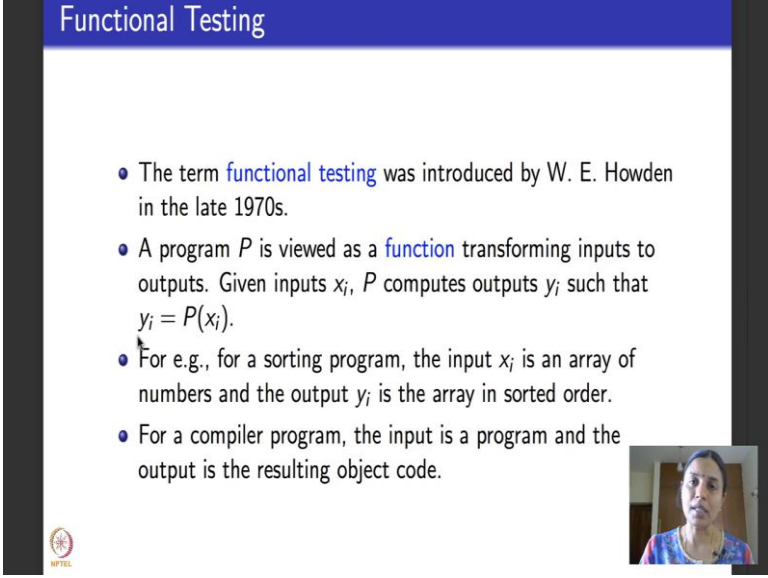
the code and whenever we were applying it to specifications, we were doing what is called black box testing.

So, black box testing involves testing of the code or a design without knowing about its internals. You do not know the language in which it is written, you do not know its structure, you do not know how many loops it has where they are. You purely look at inputs and outputs in the code and test it based purely on the expected relationship between the inputs and the outputs. Black box testing typically deals with inputs and outputs, I could apply it to code, but without looking at the structure of the code. I could apply it to design without looking at the internals of the design. I could apply to requirements by directly considering what the requirements state about the inputs and outputs.

So, this week we will fully be doing what is called black box testing. Even when we apply it to code we will purely focus only on the inputs that are there in the code, not really look at the structure of the code at all and see how to design test cases based only on the inputs. Of course, if we apply to requirements then there is no code and no need to see any structure and similarly when we apply to design, we will again focus only on the inputs and outputs.

So, we could apply to any software artifact, but remember that we are not going to design test cases based on the internal structure of the artifact. We are going to design test cases purely by looking at the inputs and the outputs of the artifact and what the artifact, which is code or design or requirements, is supposed to do while transforming the inputs to the outputs.

(Refer Slide Time: 03:02)



The slide is titled "Functional Testing" in a blue header. It contains four bullet points explaining the concept of functional testing. A small video inset in the bottom right corner shows a woman speaking.

- The term **functional testing** was introduced by W. E. Howden in the late 1970s.
- A program P is viewed as a **function** transforming inputs to outputs. Given inputs x_i , P computes outputs y_i such that $y_i = P(x_i)$.
- For e.g., for a sorting program, the input x_i is an array of numbers and the output y_i is the array in sorted order.
- For a compiler program, the input is a program and the output is the resulting object code.

So, the black box testing, if you refer to some other books popularly specially the old books, they will popularly call it as what is called functional testing. They practically mean the same. In fact, the term functional testing was introduced by this person called Howden, he was working for his statistics research institute in the late 1970s.

So, the idea is basically this. I have a program or a piece of code call it P , I will view it as a function that transforms the input that the program takes to produce outputs. So, this is a program that takes some inputs and produces outputs. When I functional test the program, I will view the program as a function that takes inputs, transforms the inputs to produce outputs.

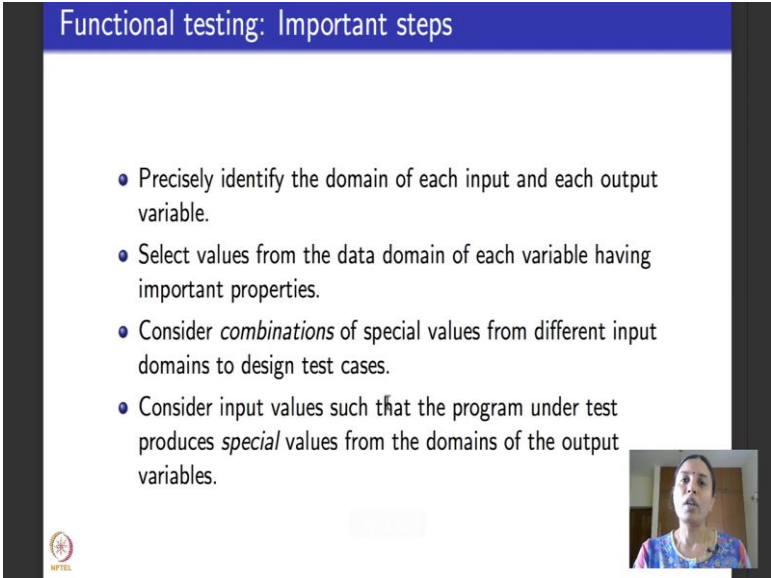
So, given a vector of inputs x_i when I say a vector of input x_i the program could have 3 inputs right like for example, it could have 2 integers and then it could have let us say a Boolean number. Another piece of program could have an array of numbers and let us say an index to search for or index whose value is to be return. So, different programs could have different inputs I put all the inputs of a program together and call it as a vector x_i . So, that is like a set of all inputs of the program that are at any given point in time, that the program is suppose to process and produce outputs.

Outputs again a given program can produce more than 1 output. So, I put all the outputs together and consider it as a vector y_i . So, what is a program? Program P can be thought of as a function that given x_i returns y_i . For example, suppose I have a program the sorts

numbers then the input x_i for the program is an array of numbers, and the output y_i is the array, but in the sorted order. So, for example, suppose another example, let us take a program which is basically a compiler, let us say a C compiler. C compiler is a program right it could be a fairly large program, but it still a program it is a piece of code what is the input to a program like a C compiler? It is another program itself.

So, I can still represent it as a fairly large input vector to a program and then what is the output to this program? The output to this program is the compiled code or in the other words ,the object code. So, the input x_i is another program and the output is the compiled or the object code.

(Refer Slide Time: 05:33)



The slide is titled "Functional testing: Important steps" in a blue header. It contains a bulleted list of four steps for functional testing. In the bottom right corner, there is a small video inset showing a woman speaking. The NPTEL logo is in the bottom left corner.

- Precisely identify the domain of each input and each output variable.
- Select values from the data domain of each variable having important properties.
- Consider *combinations* of special values from different input domains to design test cases.
- Consider input values such that the program under test produces *special* values from the domains of the output variables.

So, when I do functional testing, I consider every program as a function that takes inputs and produces outputs. So, what is functional testing? What are the important steps that people do in functional testing? They have to identify, remember I told you when we do functional testing we focus on inputs and outputs, we do not really look at the insides of a piece of program or a code.

Now, I have to understand the inputs in detail first. I have to understand what and all constitutes the input, what is the domain of each constituent in the input. So, I have to identify the domain of each input. Similarly I need to know what and all constitutes the output, what is the domain or the type of each entity in the output. So, that is the first

thing I know and understand. After I do that I select values from the domain of each input, and I select what will be the expected output for these values selected.

Several common input domains like integer, let us say typed input or a floating point typed input, let us say or an array of integers I have practically or fairly large number of values potentially infinite number of values. So, when I test for a program for the input output relationship, I cannot exhaustively test a program for all these values. The best that I can do is to select a sub set of values from the input output space. This whole week what we will be seeing is; what are the various ways in which we can select to test the program for its input output relationship.

So, this selection or the combination of selections is what this week is going to focus on. Once we select values we consider combinations of the specific values from different input domains to test, design test cases and we consider input values such that this combination that we consider for each different combination that we consider, the program we expect, produces different-different outputs.

Let us say you consider the same combination again and again for which the program produces the same output again and again, then you would not be effective in testing the program. Ideally the combinations that you select should be different in the sense that they should produce different behavior in the program, hopefully resulting in different outputs. So, that is what we will focus on.

(Refer Slide Time: 07:52)

Testing a function in context

- Even though functional testing is a black-box testing technique, sometimes, we need to know minimal *context* information to get relevant values for inputs and outputs.
- Consider a program P and a function f in P as shown in the figure below.

The diagram illustrates a program P as a box containing two components. An input x enters from the left and points to a circle inside P . A dashed arrow labeled x points from this circle to another circle labeled f . A condition $x \geq 20$ is written between the two circles, indicating a conditional execution path.

How to select and how to write combinations. Before I move on I would like to tell you that sometimes it may not be possible to purely focus on the inputs and outputs. Sometimes you might have to know a little bit about the insides of a program. Why is that so? Here is an example. This is popularly called doing functional testing in the context of something else. So, functional testing I told you is a black box testing.

So, we expect not to look at the insights of a program or a code, but sometimes we need to know a little bit. We need to know something like a minimal contextual information to be able to get relevant values for input and output. I will illustrate this through an example for you. Consider a program P, I have not really written the code for P, but assume that this is the space of program P, this is where the code of P resides. Somewhere in it is code P uses a function f. There is an input x to the program P and this x within P is passed on to call the function f whenever x is greater than or equal to 20.


So, the program P uses the input x, program P also has an internal function or a method called f. This program P uses the input x as input to the function f to call the function f whenever this condition is true. That is whenever x is greater than or equal to 20. Now let us say I want to be able to do functional testing for this program, which means what I really do not know this much that the program has a function f, that it is going to call f with x when x is greater than or equal to 20. Why I do not know about this because I am doing black box testing and I am not supposed to know. All that I know is x is an input to the program P.

So, I make an attempt to design select values for x such that I can test how the program P transforms the input x into corresponding output.

(Refer Slide Time: 09:48)

Testing a function in context, contd.

- Suppose x is a floating point variable and we are unaware of the predicate $x \geq 20$.
- We are likely to test for the following input values for x : $x = +k, x = -k, x = 0$, where k is a number with a large magnitude.
- The function f will be invoked just once, for $x = +k$.
- The valid range for input x with respect to testing f will be $x = k$ where k is a number much larger than 20, $x = y$, where $20 < y < k$ and $x = 20$.



Suppose x is a floating point variable and as I told you we are not aware of the presence of this predicate. We are not supposed to be aware, because we are doing black box testing. So, we are likely to test for the following input values of x . We will take x to be let us say one positive number, x to be one negative number and maybe we will test how the program behaves when x is 0.

I have written k as a number with a large magnitude because we are unlikely to test it for small magnitudes, because it is a floating point number you if you choose 0 then you would choose a large number and you would choose a large positive and a large negative number. Now if you see these 3 choices that we have selected for input x : $+k, -k, 0$ which of these 3 choices will actually make the program call the function f ? It will only be this $+k$ because for the other 2 choices x is $-k$ and x is 0, this predicate x greater than or equal to 20 which makes the program call the function f , that predicate becomes false.

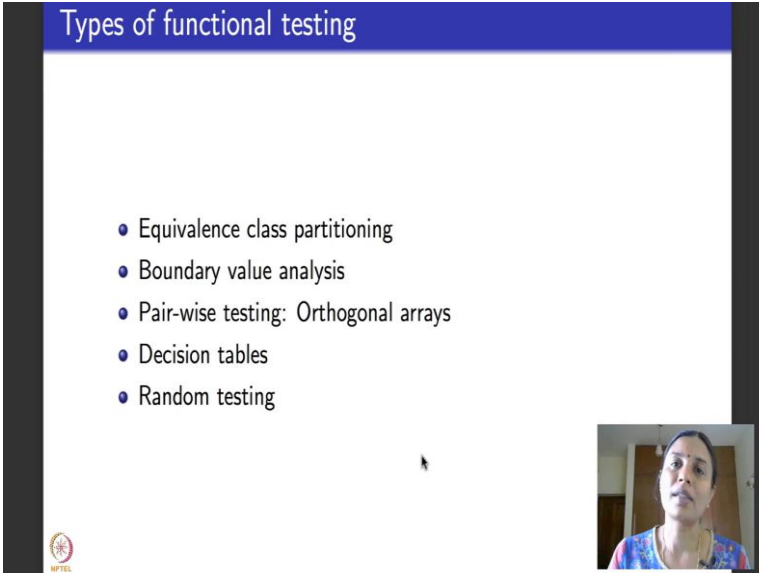
So, only for one choice I would have call this. Now suppose I want to be able to focus on functional testing of the function f , I have not really exercised f except for only once. So, suppose I know the context in which f is called. Context meaning this predicate, that x greater than or equal to 20, then I will be in little more effective in designing my test case inputs to test the function f . If I had known the context I would have designed test cases that look like this. I would say x is k , where k is a number much larger than 20 and I

would say x is y , where y is a value between 20 and k and then I would call it for x equal is equal to 20.

In all the 3 cases the function is called in this case at 20 and then in this case for a number larger than 20 and then that is in middle case for a number in between 20 and some large number, that is for a number close enough to 20. So, what I am trying to tell you through this slides is that even though we are focusing on black box testing, where we do not look at the internals of a program that looks like this, but suppose we want to do black box testing or functional testing focusing on a particular function f of the program, we might have to look at little bit inside the program just to know the context in which that function is invoked or called. Knowing the context will help us design select input values, such that we can exercise the calling of the function f itself correctly to be able to do black box testing for that function.

We still need not know the entire code for this program, it is enough to know the context in which this function is called for us to be able to help us to design good test cases.

(Refer Slide Time: 12:33)



The slide is titled "Types of functional testing" in a blue header. Below the header, there is a bulleted list of five types of functional testing. In the bottom right corner, there is a small video inset showing a woman with dark hair, wearing a colorful patterned top, speaking. The slide also features a small logo in the bottom left corner.

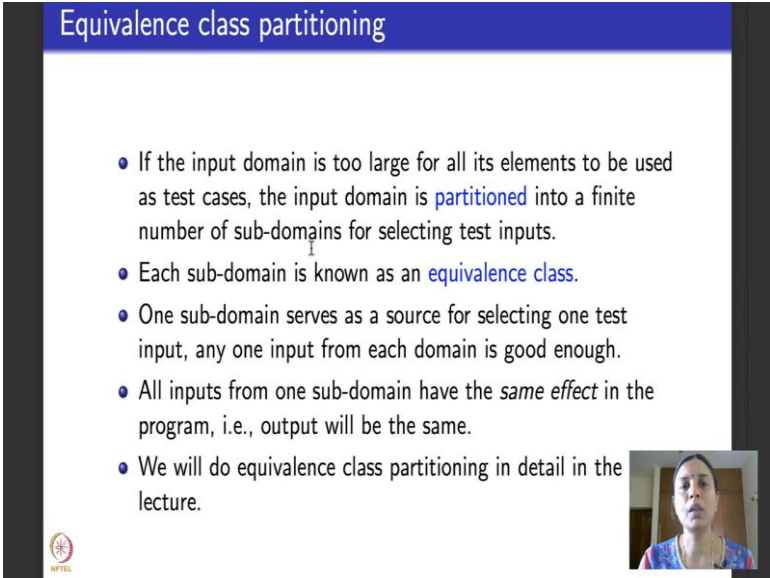
- Equivalence class partitioning
- Boundary value analysis
- Pair-wise testing: Orthogonal arrays
- Decision tables
- Random testing

What are the various types of functional testing that you will find in the testing literature? Here is a broad classification: in several books you will find this term very popular term called equivalence class partitioning or ECP or ECC, just EC. Then there is a related technique called boundary value analysis abbreviated as BVA, then you would have heard of the term pair wise testing.

In fact, there is a very popular technique which was introduced by an Indian statistician Mr. Rao called orthogonal arrays. Then you would have heard of the term decision tables and another usually popular way of doing functional testing is what is called random testing. What we will be focusing for a lot of this week is on some of these techniques, especially on equivalence class partitioning. I will call it input space portioning because that is what the text book by Amman and Offutt calls it as. Boundary value analysis we will see today. Pair wise testing we will see as one of the coverage criteria when we do equivalence class partitioning or input space partitioning. I will tell you what decision tables are today I will also tell you what random testing is today. We will do look at random testing in little more detail towards the end of the course when we do symbolic execution.

So, for the rest of this lecture, we will look at these techniques one at a time, and I will tell you through examples how to use these techniques. Equivalence class partitioning is one what we begin with.

(Refer Slide Time: 14:01)



The slide is titled "Equivalence class partitioning" in a blue header. It contains a list of five bullet points. In the bottom right corner, there is a small video inset showing a woman speaking. A small logo is visible in the bottom left corner of the slide content area.

- If the input domain is too large for all its elements to be used as test cases, the input domain is **partitioned** into a finite number of sub-domains for selecting test inputs.
- Each sub-domain is known as an **equivalence class**.
- One sub-domain serves as a source for selecting one test input, any one input from each domain is good enough.
- All inputs from one sub-domain have the *same effect* in the program, i.e., output will be the same.
- We will do equivalence class partitioning in detail in the lecture.

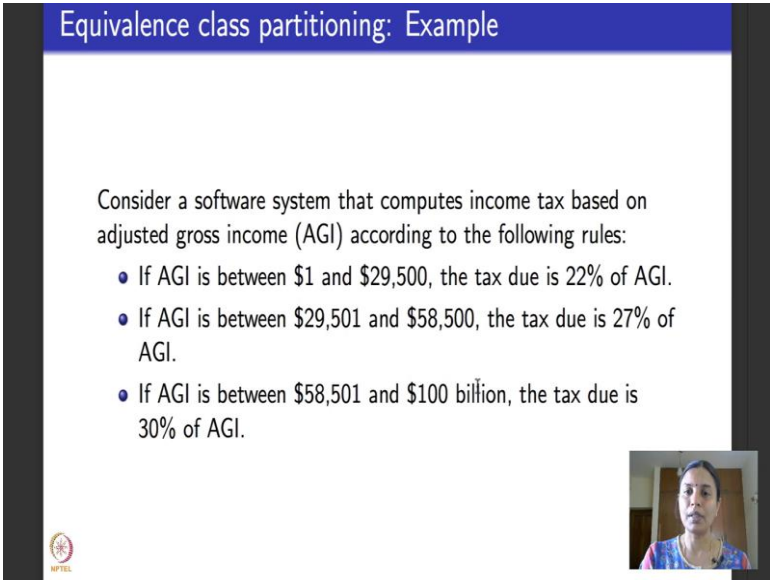
So, equivalence class partitioning basically assumes that the input domain that I want to test is fairly large. Let us say the input is a floating point number, the input is an integer so it is practically impossible to test the program for all values of the inputs. So, I want to be able to partition the set of inputs such that the number of partitions is finite, and each such partition is known as an equivalence class. The word equivalence class comes

because as far as the program is concerned, it is expected to behave similarly for each element that chosen from the partition.

So, we will make this very precise and clear. I will define it to you what a partition is and how to select inputs from partitions, what are the properties that partitions should satisfy everything in the next lecture. But for now equivalence class partitioning means you take an input domain it is a fairly large set, I partition the input domain into several sub sets, a finite number of them such that I pick one input value from each partition. And the inputs have the property that from each partition if I pick one input value then the program when it runs on that randomly picked input, will produce the same output as any other input picked from the same partition.

So, as I told you we will do equivalence class partitioning as input space partitioning in detail in the next lecture.

(Refer Slide Time: 15:32)



The slide is titled "Equivalence class partitioning: Example" in a blue header. The main content area is white with black text. It starts with "Consider a software system that computes income tax based on adjusted gross income (AGI) according to the following rules:" followed by a bulleted list of three rules. In the bottom right corner, there is a small video inset showing a woman speaking. A small logo is visible in the bottom left corner of the slide area.

Equivalence class partitioning: Example

Consider a software system that computes income tax based on adjusted gross income (AGI) according to the following rules:

- If AGI is between \$1 and \$29,500, the tax due is 22% of AGI.
- If AGI is between \$29,501 and \$58,500, the tax due is 27% of AGI.
- If AGI is between \$58,501 and \$100 billion, the tax due is 30% of AGI.

So, here is a small example that motivates equivalence class partitioning. So, remember I am doing black box testing. So, I do not look at the program at all, I will just look at the requirements or what the program is supposed to do. So, here is some program that to calculates income tax for some country. So, here are the rules based on which the program is supposed to calculate the income tax. To calculate income tax the program considers an input called adjusted gross income abbreviated as AGI and it says if the

adjusted gross income is between 1 and 29500 then the amount of tax that is payable by that person is 22% of the AGI.

If the AGI is between 29501, that is it exceeds this 29500, but is within 58500 then the amount of tax to be paid by that individual is 27% of the AGI. If the AGI exceeds 58500 dollars, that is it is at least 58501 and if it is less than 1 billion which is a lot of money, then the tax to be paid by the individual is only 30% of the AGI. So, these are some rules it says based on these rules you take the input as AGI, the annual gross income and apply these rules to calculate the amount of tax to be paid by an individual.

So, that is a program that is as it and we are doing black box testing for that program. So, we do not get to see the program, we do not get to look at, we have to still be able to design test cases purely based on this information. What is the information that is given in the slide the program has one input AGI, the program is suppose to produce one output which is the amount of tax to be paid. Here are the rules if AGI is between 1 and 29500 tax is 22 %. If AGI is between 29501 and 58500, tax is so much. If AGI is above 58501 and less than 1 billion then the tax is so much. We know this much, so we know inputs, we know outputs, and we know how the program is expected to compute the output given input.

Now, if you see this input there could be so many individuals who have different-different AGIs, I clearly cannot test the program by giving each individual number as annual gross income and test and it is not necessary also. Why is not necessary, because if you see the requirements here it clearly states that if the annual gross income is between one dollars and 29500 dollars, which means for any value between these 2 numbers the tax is always 22 % of that value. So, it is enough to pick up one value from this number to be able to test it right.

And then, similarly it is enough to take up one value between the second range to test it, is enough to pick up one value between the third range to test it, and we also typically pick up values that are outside the range.


(Refer Slide Time: 18:47)

Equivalence class partitioning: Example, contd.

We get five partitions as below:

- $1 \leq \text{AGI} \leq 29,500$: Valid input.
- $\text{AGI} < 1$: Invalid input.
- $29,501 \leq \text{AGI} \leq 58,500$: Valid input.
- $58,501 \leq \text{AGI} \leq 100 \text{ billion}$: Valid input.
- $\text{AGI} > 1 \text{ billion}$: Invalid input.

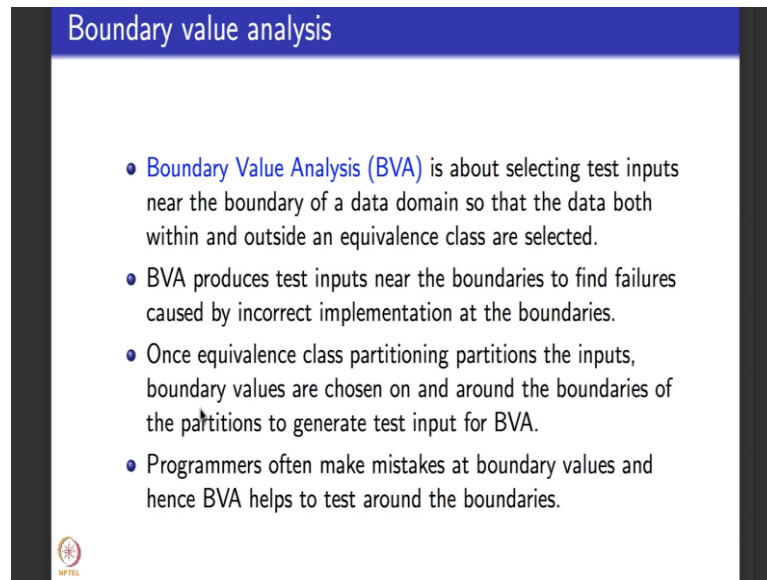
Five test cases, each containing one number for AGI in the above range will suffice for testing the tax requirement based on AGI.



So, I will give a number less than 1 dollars and I will give one number greater than 1 billion which constitute the boundary of these equivalence class partition. So, totally we get 5 partitions: one partition is for between one, annual gross income between one and 29500, it is a valid input, another partition is for the annual gross income being less than 1, it is an invalid input, because annual gross income is a negative number let us say or let us say 0.

The third partition which is again valid is annual gross income it is between 29501 and 58500, the fourth partition which is another valid input it is annual gross income is between 58501 and 100 billion and there one final outside the value or invalid input, which is annual gross income will greater than 1 billion. Maybe that person does not have to pay any tax. So, 5 test cases one for this value one for this range, one for this range, one for this range and one for this range is enough if fully test the program for correct functionality.

(Refer Slide Time: 19:41)



The slide is titled "Boundary value analysis" in a blue header. It contains a bulleted list of four points. The first point defines BVA as selecting test inputs near the boundary of a data domain. The second point states that BVA produces test inputs near the boundaries to find failures. The third point explains that once equivalence class partitioning is done, boundary values are chosen on and around the boundaries of the partitions. The fourth point notes that programmers often make mistakes at boundary values, so BVA helps to test around them. A small logo is visible in the bottom left corner of the slide.

- **Boundary Value Analysis (BVA)** is about selecting test inputs near the boundary of a data domain so that the data both within and outside an equivalence class are selected.
- BVA produces test inputs near the boundaries to find failures caused by incorrect implementation at the boundaries.
- Once equivalence class partitioning partitions the inputs, boundary values are chosen on and around the boundaries of the partitions to generate test input for BVA.
- Programmers often make mistakes at boundary values and hence BVA helps to test around the boundaries.

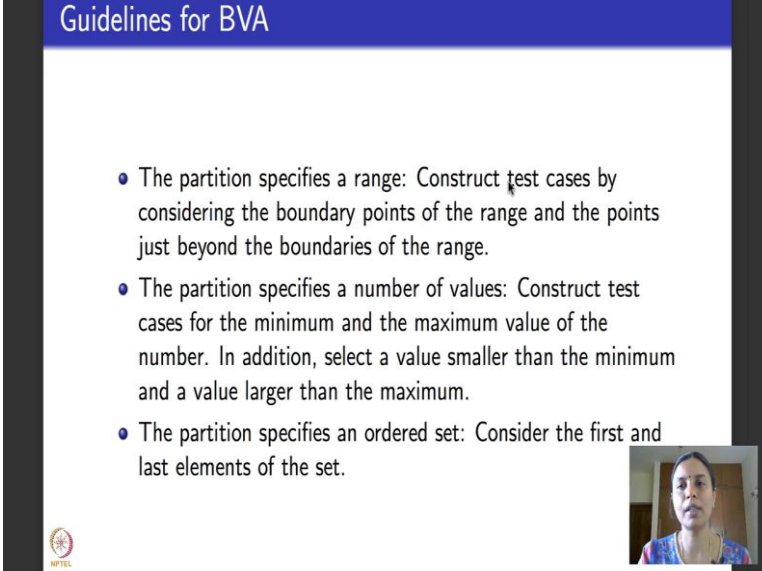
Moving on, there is another popular technique in functional testing called boundary value analysis abbreviated as BVA. What is BVA about? It is about selecting test inputs nearer the boundary of a data domain so that the data within and outside the boundary is well taken care of. For example, if you go back to the same example, what constitutes boundaries? Boundaries are what if the AGI is one rupee, one dollar sorry, what if the AGI is exactly 29500 dollars, what if the AGI is let us say 29500.5 dollars it is still more than this, but it is less than this right.

So, boundaries values constitute the boundaries of the various partitions and the idea is to be able to use the values at the boundaries to be able to produce test inputs, because usually programmers makes small mistakes there or people who writes specification makes small mistakes there. They would not have taken care of the boundary values correctly, they would not have put the less than or equal to when they had to put it. They would may be put less than instead of less than or equal to. So, it is useful to pick up input values exactly at and around the boundary to be able to functional test a program and boundary value analysis is about doing that.

Once equivalence class partitioning partitions the input, we choose boundary values so that they are on or around the boundaries of the partition to generate test inputs for BVA. Why, because we believes that typically programmers make mistake at the boundaries and those errors are easier to catch and the test inputs design for BVA will help us to

catch those errors. So, how do you do boundary value analysis. Here are the guidelines: the equivalence class partitioning specifies a range like we saw here right, here is a range first range what we one and 29500 second range says less than 1 and so on.

(Refer Slide Time: 21:46)



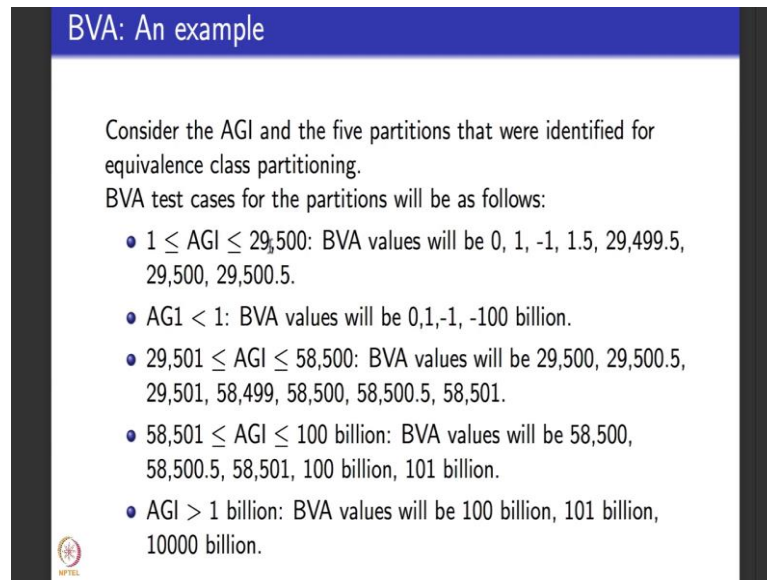
The slide is titled "Guidelines for BVA" in a blue header. It contains three bullet points, each preceded by a blue circular icon. The first bullet point discusses constructing test cases for a range by considering boundary points and points just beyond the boundaries. The second bullet point discusses constructing test cases for a number of values by considering minimum and maximum values, as well as values smaller than the minimum and larger than the maximum. The third bullet point discusses constructing test cases for an ordered set by considering the first and last elements. In the bottom right corner of the slide, there is a small video inset showing a woman with dark hair, wearing a patterned top, speaking.

- The partition specifies a range: Construct test cases by considering the boundary points of the range and the points just beyond the boundaries of the range.
- The partition specifies a number of values: Construct test cases for the minimum and the maximum value of the number. In addition, select a value smaller than the minimum and a value larger than the maximum.
- The partition specifies an ordered set: Consider the first and last elements of the set.

So, you take that range and you construct test cases by considering the boundary point of the range, and the points just beyond the boundary of the range and if the partition specifies number of values, we will see examples of this as we move on, you construct test cases for the minimum value and for the maximum value as you mean that the values are order.

If the partition specifies an ordered set then you consider the first and the last element of the value. This is how you pick up test case input values while you are doing boundary value analysis.

(Refer Slide Time: 22:13)




BVA: An example

Consider the AGI and the five partitions that were identified for equivalence class partitioning.

BVA test cases for the partitions will be as follows:

- $1 \leq \text{AGI} \leq 29,500$: BVA values will be 0, 1, -1, 1.5, 29,499.5, 29,500, 29,500.5.
- $\text{AGI} < 1$: BVA values will be 0, 1, -1, -100 billion.
- $29,501 \leq \text{AGI} \leq 58,500$: BVA values will be 29,500, 29,500.5, 29,501, 58,499, 58,500, 58,500.5, 58,501.
- $58,501 \leq \text{AGI} \leq 100 \text{ billion}$: BVA values will be 58,500, 58,500.5, 58,501, 100 billion, 101 billion.
- $\text{AGI} > 1 \text{ billion}$: BVA values will be 100 billion, 101 billion, 10000 billion.



So, we will take the same annual gross income and tax computing based on the annual gross income example. If you remember we had defined 5 partitions here is it these 5 partitions. What will BVA do? This was the first partition right one less than or equal to AGI less than or equal to 29500 boundary value analysis will pickup values around the boundary. So, let us take the left boundary here left boundary is one less than or equal to.

So, the values around the left boundary will be 0, 1 which is at the boundary, minus 1 which is before the boundary 1.5 which is just after the left boundary. Similarly for the right boundary what we do is we take 29500 and 99.5 which is just below the right boundary 29500 at the boundary 29500.5. Why did I choose this 0.5, mainly because if you see the next one assumes that it is 29501. So, let us say the annual gross income is a floating point number then the program should be well take up to handle 29500.5, may be you should over approximate it to be 29501 and then calculate tax, but the programmer should have taken care of it.

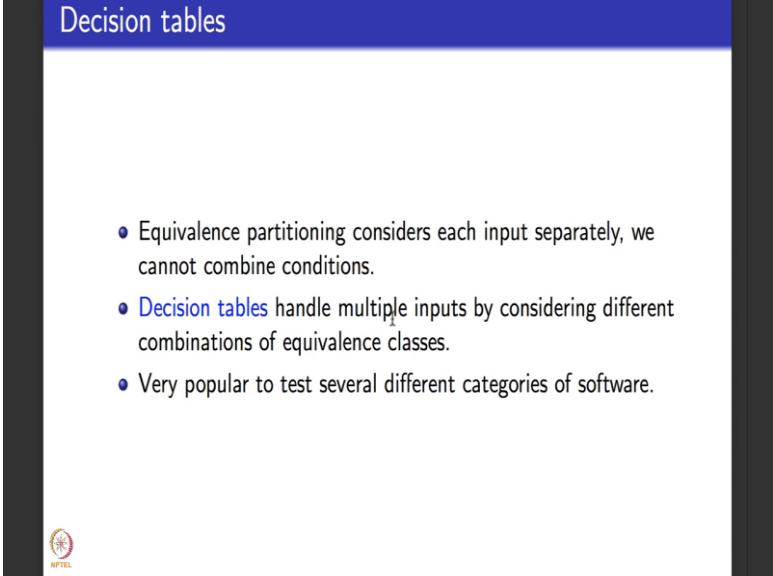
So, writing a test case like this will help to find an error if the program is not taking care of it. Now moving on, for the second partition which was this annual gross income less than 1, boundary value analysis will be 0, 1, -1 and something like -100 billion, which is like a fairly large negative number. -100 billion if you do not like it, you put it as -3, -7, it will be the same as for as the program behavior is cons. Again for the third partition you take this which is a third partition that was given in the equivalence class

partitioning right boundary values 29500, 29500.5, 29501 which tests for the left boundary. For the right boundary you have 58499, 58500, 58500.5, 58501, this tests for the right boundary.

Similarly, if a next partition right values around the boundary the last partition right values around the boundary here the last value that I have written here I had given some fairly large number just to make sure that the program handles some corner cases, but if you do not like this fairly large number you could give something like 100 and 5 billion or something like that and stop it. So, is it clear?

Please, how BVA works? BVA assumes that equivalence class partitioning or input space partitioning has been done on the program and designs test cases by picking up input values on and around the boundary of each partition.

(Refer Slide Time: 24:51)



The slide has a blue header with the text "Decision tables". Below the header, there are three bullet points. The second bullet point, "Decision tables handle multiple inputs by considering different combinations of equivalence classes.", is highlighted in blue. In the bottom left corner, there is a small circular logo with a sun-like symbol and the text "NPTEL" below it.

- Equivalence partitioning considers each input separately, we cannot combine conditions.
- **Decision tables** handle multiple inputs by considering different combinations of equivalence classes.
- Very popular to test several different categories of software.

The next popular functional testing technique that you would have heard of it is one of the most popular functional testing techniques that I know of is what is called decision tables. Why do decision tables come into picture? They come into picture for the following reason. A technique like equivalence class partitioning considers each input separately, it cannot handle a combination of inputs. In fact, in the example that we saw which involved annual gross income, there is exactly one input which is the annual gross income.

So, suppose there was an annual gross income and age and the tax rule says that if the annual gross income is so much and the age is above 60, let us say the person is a senior citizen then the tax is slightly less. Equivalence class partitioning may not be able to handle it for that you need what is called decision tables.

(Refer Slide Time: 25:48)

Decision table									
Conditions	Values	Rules or Combinations							
		R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8
C_1	Y, N, -	Y	Y	Y	Y	N	N	N	N
C_2	Y, N, -	Y	Y	N	N	Y	Y	N	N
C_3	Y, N, -	Y	N	Y	N	Y	N	Y	N
Effects									
E_1		1		2	1				
E_2			2	1			2	1	
E_3		2	1	3		1	1		
Checksum	8	1	1	1	1	1	1	1	1

Decision tables can handle multiple inputs by considering different combinations of equivalence classes. So, how does a decision table look like. This is how it will look like. There will be a fairly large table. What do each of these things mean, we will see. So, what are the entities that are there in the decision table. There are something called conditions, then there are things called values. Values are typically yes abbreviated as y, n abbreviated as no, I mean sorry no abbreviated as n, and this dash rewrite as do not care. What do I mean by do not care? Do not care means, it is the value is immaterial of whether it is yes or no and these rules are basically populated with yes, no and do not care symbols for each condition.

So, it is says condition one should be satisfied, that is what this yes mean in rule one. Condition 2 a condition one should be satisfied in rule 2, condition 2 should not need not be satisfied in rule 3. So, that is why it is written as a no. So, these rules basically say for each of the conditions whether each condition is needed ,not needed or I do not care. In this particular example I have just filled it up with yeses and nos I have not given do not cares, but you can write do not cares also. Then the later half of the decision table has

what a called effects. Let us say here there are 3 effects E 1, E 2, E 3 and then there is a checksum which tells you what happens to each of these effects.

(Refer Slide Time: 27:19)

Decision table

A decision table has

- A set of **conditions** and a set of **effects** arranged in a column.
- Each condition has possible value (yes (Y), no(N), don't care(-)) in the second column.
- For each combination of the three conditions there are a set of rules from R_1 to R_8 . Each rule has a Y/N/- response and contains an associated set of effects (E_1 , E_2 and E_3).
- For each effect, an **effect sequence number** specifies the order in which the effect should be carried out if the associated set of conditions are satisfied. For e.g., if C_1 and C_2 are true but C_3 is not true, then E_2 should be followed by E_3 .
- The **checksum** is used for verification of the combinations the decision table represents.

So, how do I read it? I read these as do it in order of priority, means if, we will see an example. So, what is a decision table have it has a set of conditions and a set of effects which are arranged in a column like this, the first column. Each condition has possible values yes no do not care, in the second column that is what is written here. For each combination of the 3 conditions there are rules, in this case there are 8 rules, it need not be 8 rules all the time rules are flexible, you can write as many as you want to be able to exhaustively capture the conditions on the inputs. Each rule can be a yes no or a do not care and with each rule, there are a set of effects.

Like for example, with rule one which are the 2 effects that are associated with rule one effect one E 1 and effect 3 E 3, effect 2 is left blank which means effect 2 does not matter for rule one and what is this order say, 2 and 1? It is says if this is true which means all the 3 conditions are satisfied in which case rule one applies effect one occurs first and then effect 2 occurs. So, for each effect there is a sequence number which is what is this part, effect sequence number which specifies the order in which the effect should be carried out if the associated set of conditions are satisfied. For example, if C 1 and C 2 are both yes and C 3 is a no, then E 2 should be followed by E 3.

So, let us see if C 1 and C 2 are both yes, which is rule 2, and C 3 is a no, then E 3 is followed by E 2, is it clear. So, what is checksum? Checksum is used to the standard use of checksum is to verify if the combination of the values is correct. So, this is how a decision table looks like.

(Refer Slide Time: 29:04)

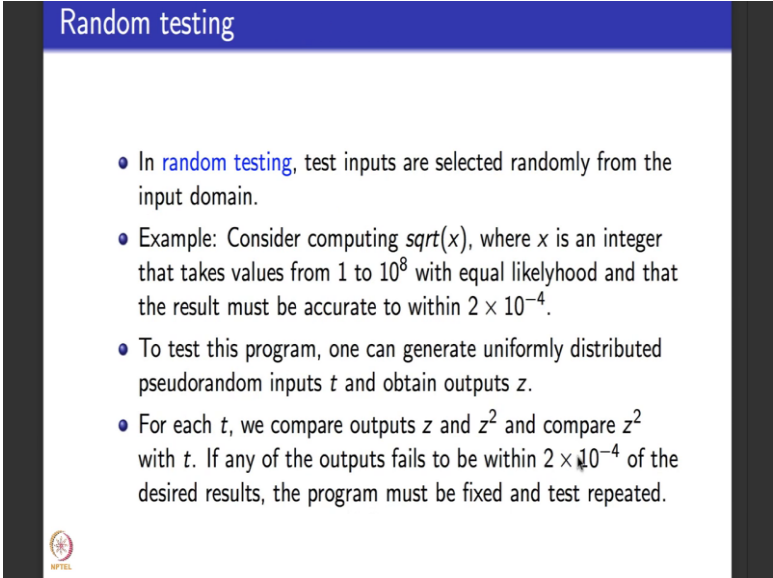
Decision table: Example				
Conditions	Step 1	Step 2	Step 3	Step 4
Repayment amount has been mentioned	Y	Y	N	N
Terms of loan has been mentioned	Y	N	Y	N
Effects				
Process loan amount	Y	Y	–	–
Process term	Y	–	Y	–
Error message	–	–	–	Y

So, we will see a small example. So, let us say there is a loan management system and that calculates what is the amount of loan taken what is the amount to be repaid and what are the what is the term is to be paid over 3 years, over 5 years, over 15 years what is it rates. So, there are 2 kinds of inputs repayment amount is one input term of a loan which is a number of years with which we repay the loan is another input. So, the condition says, has the repayment amount been mentioned, yes or no? Similarly as the term of the loan been mentioned yes or no and what is the effect? If the repayment amount has been mentioned and the term of loan has been mentioned, then you can process the loan amount and you can process the term. But let say if the repayment amount has been mentioned and the term of loan is not been mentioned, that it is a no, then you can only process the loan amount you cannot process the term.

Similarly if the amount is not mentioned, but the term has been mentioned then you cannot process the loan amount you can only process the term. If both have not been mentioned that is, it is a no then you have to give an error message. So, this is how you write requirements for a design decision table. Based on these requirements you write

test cases. So, if these rules are true in this case, you write a test case that will do processing now by taking input as repayment amount taking input as the term of loan and produce the expected output.

(Refer Slide Time: 30:47)



The slide is titled "Random testing" in a blue header. It contains four bullet points:

- In **random testing**, test inputs are selected randomly from the input domain.
- Example: Consider computing $\text{sqrt}(x)$, where x is an integer that takes values from 1 to 10^8 with equal likelihood and that the result must be accurate to within 2×10^{-4} .
- To test this program, one can generate uniformly distributed pseudorandom inputs t and obtain outputs z .
- For each t , we compare outputs z and z^2 and compare z^2 with t . If any of the outputs fails to be within 2×10^{-4} of the desired results, the program must be fixed and test repeated.

There is a small logo in the bottom left corner of the slide.

The second column, it will take input only as the repayment amount, the third column, it will take input only as the term of processing. In forth column it will it just handles error handling is the program done exception handling or error handling well that is what the forth case will test in the decision table. I hope decision tables are clear. Moving on we will bind up this module with brief look at random testing. What does the word random say? Random says pick a random input and test. You might wonder why is this useful. In fact, it turns out to be a very popular testing technique. There is a popular class of tools called monkey runners which is like imagine monkey producing inputs to a program.

So, this tool will produce inputs that are as random as that. These are very useful when you test programs if it has handled exceptions and all kinds of error conditions very well. So, random testing basically, it selects a test input randomly from the input domain and gives it to a program. But they also see how do we use randomly selected inputs to know if a program has an error, here is an example that illustrates that. Consider a program that computes this square root of a number I have written it like this $\text{sqrt}(x)$, where x is an integer. Let us say x takes values from 1 to 10^8 with equal likelihood, and we say that the result this program produces must be accurate up to within this number 2×10^{-4} .

So, to test this program we can generate a uniformly distributed pseudo-random input, let us say t , and let us say the program produces output z which is supposed to be \sqrt{t} . So, for each such t what do we do? We take z and we take z^2 . So, if z is actually the \sqrt{t} then z square should be close enough to t . In fact, it should be within this likely permissible range 2×10^{-4} . If z^2 fails to be within this range of t then the program has an error so that must be fixed and then this test can be repeated. So, this is how random testing happens. So, random testing may or may not be effective all the time. Later in the course we will see one very effective way of overcoming random testing called symbolic testing.

So, this module's goal was to give you an overview of the various functional testing techniques. From next class onwards, I will focus only on equivalence class partitioning or called input space partitioning. We will see how to define partitions of a set, how to design input space partitions for various kinds of programs, and what are the coverage criteria based on these partitions.

Thank you.

Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 32
Input Space Partitioning

Hello again. Welcome to the third lecture of week 3. What did we do last time? If you remember I had introduced you to functional testing, how is functional testing done popularly? And we looked at various techniques of functional testing, an overview of functional testing. In that one of the testing techniques was equivalence class partitioning.

Today I had like to spend some time on that. We call equivalence class partitioning as input space partitioning because it involves partitioning the input domain. So, it is practically the same as equivalence class partitioning. So, I will introduce you to that domain of equivalence class partitioning or input space partitioning in detail today.

(Refer Slide Time: 00:49)

Partitions of a set

- Given a set S , a **partition** of S is a set $\{S_1, S_2, \dots, S_n\}$ of subsets of S such that
 - The subsets S_i of S are **pair-wise disjoint**, i.e., $S_i \cap S_j = \emptyset$.
 - The union of the subsets S_i is the entire set S , i.e., $\cup_i S_i = S$.

S

The diagram shows a large rectangle labeled S at the top. Inside this rectangle, the space is divided into four regions labeled S_1 , S_2 , S_3 , and S_4 . S_1 and S_2 are on the left, S_3 and S_4 are on the right. The boundaries between these regions are curved lines that meet at a central point, ensuring that the regions are disjoint and their union is the entire set S .

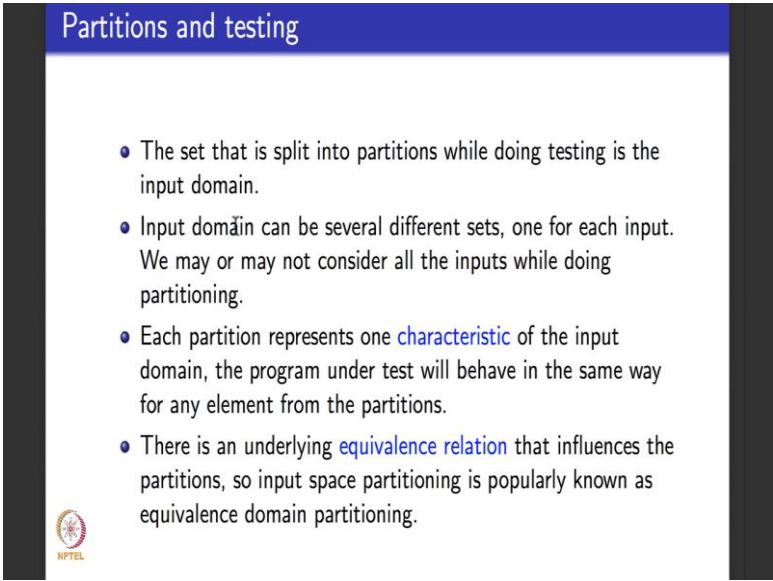
So, we begin by understanding what are partitions of a set. So, I am given a set S , set S could be a finite set, an infinite set, a countable finite set, we really do not worry about what kind of a set it is. I want to understand what the partitions of the set are. So, partition of a set is a collection of subsets of the set S . Here it is a collection of n subsets such that each subset is disjoint pair-wise with the other one that is they are mutually

disjoint. S_i intersection S_j is empty, they do not have any elements in common and put together the union of all the subsets you get back the entire set. So, that intuitively explains the word partition also.

Partition means I take a set S divide it into various subsets, the various subsets that I divide it into should satisfy a few properties. The number of subsets that I divide it into should be finite, in this case it is n . Each of these subsets should not have anything in common with the any other subset that is they should be pair wise disjoint and put together all the subsets together constitute the entire set S . Nothing is left out after partitioning the set S .

So, in this small figure here, the set S is partitioned into 4 partitions: S_1 , S_2 , S_3 and S_4 . If you see put together these 4 partitions constitute the entire set and they are pair wise disjoint. So, this is small example will illustrate how partitions work.

(Refer Slide Time: 02:21)



The slide has a blue header with the text "Partitions and testing". Below the header is a white box containing a bulleted list. At the bottom left of the white box is a small circular logo with the word "NPTEL" underneath it.

- The set that is split into partitions while doing testing is the input domain.
- Input domain can be several different sets, one for each input. We may or may not consider all the inputs while doing partitioning.
- Each partition represents one **characteristic** of the input domain, the program under test will behave in the same way for any element from the partitions.
- There is an underlying **equivalence relation** that influences the partitions, so input space partitioning is popularly known as equivalence domain partitioning.

Now, how are we going to use partitions as far as testing is concerned. So, the set, this set S that I defined for a partition of S the set that we are going to partition is what is the input domain. So, take a program, the program will have several inputs all the inputs could be a various types. The input domain as we discussed in the last lecture constitutes all the types of the inputs that the program has along with the domain types as declared within the program. It can be thought of as a Cartesian product or a cross product of all the input types. For example, if a program has a let us say two Boolean inputs and let us

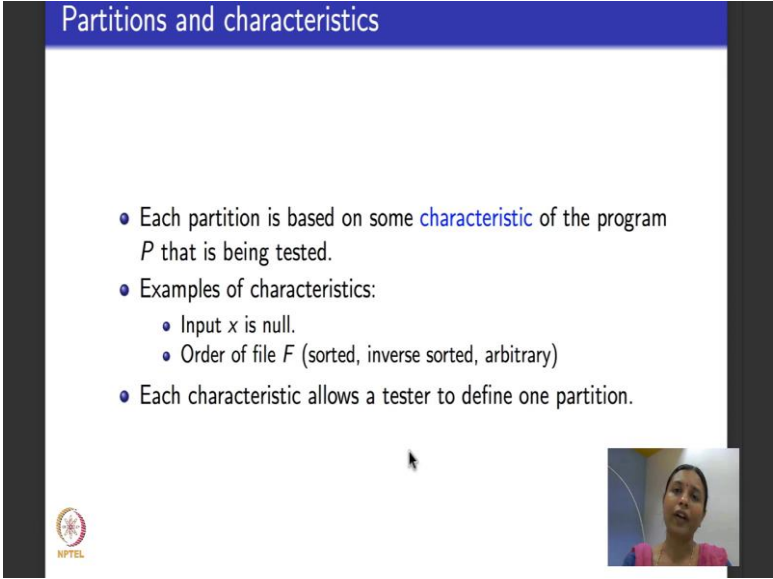
say one integer input, then the input domain will be the set $\{0, 1\}$ for one Boolean input cross product-ed at with $\{0, 1\}$ for the second Boolean input, cross product-ed with the domain of integers based on the kind of computer the program is going to run on.

So, that is the set that we are going to take and partition. Input domain as I told you can be several different sets typically one for each input, while doing partitioning we may or we may not consider all the inputs. Testers typically sometimes decide to focus only on certain inputs do not consider all the inputs for partitioning, but when they are doing let say integration level testing, they might consider all the inputs for partitioning.

So, it widely varies based on what is being tested and the testers choice. It is believed that each partition of an input domain represents one property or in the testing jargon we call it characteristic. It represents one characteristic of the input domain and the program that is being tested will behave in exactly identical way for any input value from each partition. So, I take as input domain, partition it into several partitions and pick one value to define my test case from each partition, and the belief is the program behavior in terms of the expected out to the program is expected to produce, will be exactly the same if I pickup any input from each partition. But inputs across partitions, the program is expected to behave differently because each partition represents a different characteristic.

And there is an underlying equivalence relation. Equivalence relations are binary relations that satisfy certain properties. We are not interested in defining or understanding what they are, but equivalence here means that the behaviour of the program is identical for any choice of input from a given partition. So, we say that the set of partitions are equivalence classes induced by an equivalence relation.

(Refer Slide Time: 04:53)



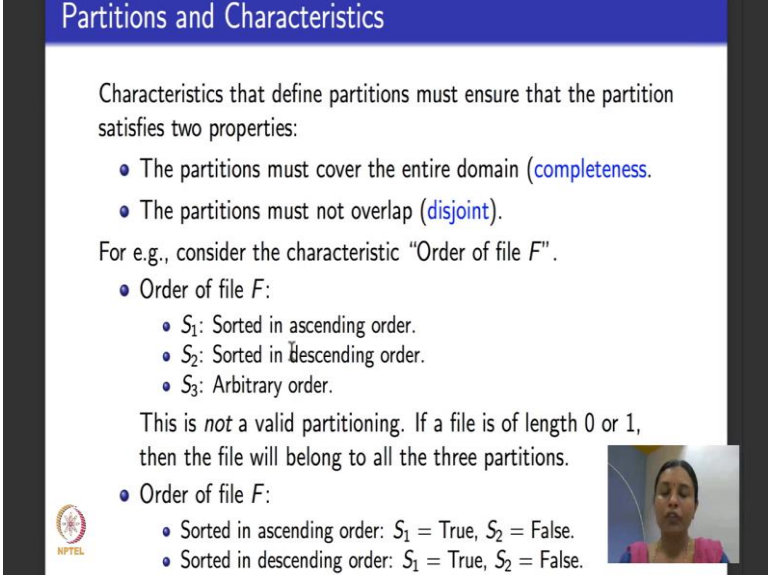
The slide has a blue header with the title "Partitions and characteristics". Below the header, there is a white area containing a bulleted list. In the bottom right corner of the slide, there is a small video inset showing a woman speaking. In the bottom left corner, there is a small NPTEL logo.

- Each partition is based on some **characteristic** of the program P that is being tested.
- Examples of characteristics:
 - Input x is null.
 - Order of file F (sorted, inverse sorted, arbitrary)
- Each characteristic allows a tester to define one partition.

So, how do we define partitions and characteristics? Each partition as I told you is based on a characteristic or some property of input domains. So, here are some example properties or characteristics. You could be worried about a property which asks whether the input is null or not, you could be worried, let us say the input is a file, you would be worried about whether the file is in sorted order or not and if it is in sorted order, what is the order of sorting, is the file in sorted order is it in inverse sorted order or it is an arbitrary order, it is not sorted at all.

So, each characteristic allows a tester to define one partition. So will see a few examples that explain how to define additional characteristics.

(Refer Slide Time: 05:29)



Partitions and Characteristics

Characteristics that define partitions must ensure that the partition satisfies two properties:



- The partitions must cover the entire domain (**completeness**).
- The partitions must not overlap (**disjoint**).

For e.g., consider the characteristic "Order of file F ".

- Order of file F :
 - S_1 : Sorted in ascending order.
 - S_2 : Sorted in descending order.
 - S_3 : Arbitrary order.

This is *not* a valid partitioning. If a file is of length 0 or 1, then the file will belong to all the three partitions.

- Order of file F :
 - Sorted in ascending order: $S_1 = \text{True}$, $S_2 = \text{False}$.
 - Sorted in descending order: $S_1 = \text{True}$, $S_2 = \text{False}$.

So, what are we looking at, we are looking at two properties as I told you if you remember in the first slide I told you what do partitions satisfy? They have to be pairwise disjoint and put together they have to constitute the entire set. So, when we look at characteristic, we are looking at characteristics, the partitions induced by the characteristics must cover the entire domain, put together they must cover the entire input domain. In testing terminology we call it the partitions are complete or completeness property and the disjointness should still hold, pairwise partitions must never overlap, every pair of partitions should be disjoint with the other one.

So, for example, if I consider the characteristic that I showed you in this slide the order of a file: is it sorted is it inverse sorted or is it arbitrary. Here is one example of an attempt to define a characteristic and go ahead and look at partitions based on that characteristic. So, I could say, here are the 3 direct partitions, first partition could ask whether the file is sorted in ascending order, second partition could be all files that are sorted in inverse ascending or descending order, third partition could be all files that are not sorted that is they are in arbitrary order.

But remember if this is the partitioning and we want to go ahead and check whether these partitioning satisfy these properties or not. If the partitions do not satisfy the properties of completeness and disjoint we do not work with them for testing at all, because testing without these two properties can be considered to be have errors, it could

have lot of issues and problems. So, partitions have to be complete, they have to be disjoint.

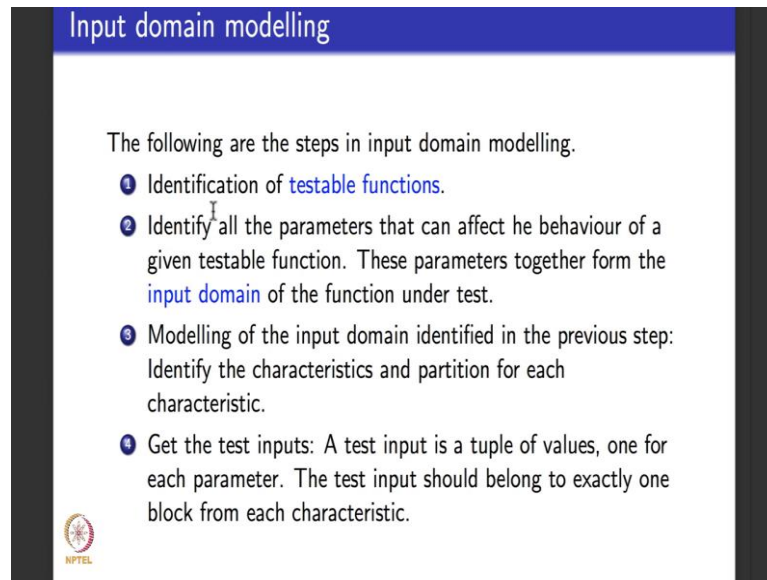
So, here for this example partition, is it complete is it just joint? The problem is it is not disjoint. Why is it not disjoint? It is not disjoint because suppose I consider small files that is files that have length 0 or length 1, then the file will be sorted ascending, the file will be sorted descending and the file will can be thought of as not being sorted at all. So, such files that are of length 0 or length 1 will belong to all the 3 partitions, which means the disjointedness property is not satisfied.

So, what do we do? What we usually do is we consider the same characteristic, but not in exactly this way. We tweak it a little bit and write it is likely differently. So, we write the characteristic of the order of the file as follows. We ask a question for the characteristics. So, we ask is the file sorted in descending order or is the file sorted in ascending order? The answer could be yes in which case it is true and the answer could be no in which case it is false.

So, if I write the same thing like this, you will observe that the disjointedness problem goes away. How does it go away? Take a file of length 0 or length 1. So, it will be sorted in ascending order. So, it belongs to the true part of this partition, it will be sorted in descending order, so, it will belong to the true part of the second partition. But the first characteristic which is been sorted in ascending order induces two partitions: is it true is it false and a file of length 0 or 1 belongs to exactly one of these, and the second characteristic which is the characteristic of being sorted in descending order induces two more partitions: true and false and a file of length 0 or 1 again belongs to exactly one of these partitions.

So, we basically consider partitions the way intuition tells us by looking at properties by looking at specifications, but what is what I am trying to tell you through this slide is that you should be a little careful in way you write it. If you write it directly like this sorted ascending, sorted descending, sort it arbitrary then the partitions may not be valid, but if you consider each one individually and consider it is being true or false and being true or false then the partitions satisfy the disjointedness and completeness property. So, they are valid partitions.


(Refer Slide Time: 09:45)



Input domain modelling

The following are the steps in input domain modelling.

- 1 Identification of testable functions.
- 2 Identify all the parameters that can affect the behaviour of a given testable function. These parameters together form the input domain of the function under test.
- 3 Modelling of the input domain identified in the previous step: Identify the characteristics and partition for each characteristic.
- 4 Get the test inputs: A test input is a tuple of values, one for each parameter. The test input should belong to exactly one block from each characteristic.



Will move on to considering the input domain and how to model the input domain. What do we mean by modelling the input domain? Modelling the input domain means that I have to be able to understand which are the sets that constitute the input domain for my consideration to be testing now. And how do I go ahead and partition, identify characteristics for these sets and partition them based on these characteristics. Typically what do we do? Here are the steps that we undertake.

The first step that we do is we identify testable functions. When is a function testable? If you remember in the last lecture I had shown you an example of knowing the context right there was a large program p , that was calling a particular function right. So, that function inside the program p was the function under test when we were doing input space partitioning or equivalence partitioning for that functions. So, in general program could have several different testable functions.

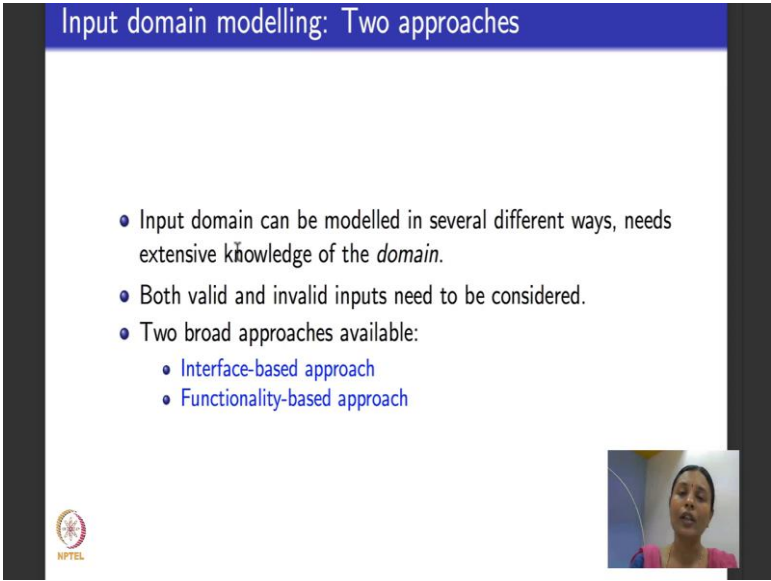
So, we now focus on which are that functions that we want to begin testing on. So, that process is what is called identifying testable functions. So, I begin by identifying which are the functions I want to test. So, those are what are called testable functions. The second step is to identify all the parameters that can affect the behaviour of a testable function. What like for example, if you remember in the previous lecture that particular function f was called only when x was greater than 20 and that property was very important for that. I called it as test in the function in context, that is what this step is

talking about. Once you identify all the parameters these parameters put together constitute the input domain of the function that is being tested.

Now, once I have the input domain I have to be able to model it. Now what do I do here? I look at the properties of the function under test. what do I want to test about this function that is under test? What are the requirements that this function is suppose to satisfy, what is it is functionality? I look at all that, that will help me to identify characteristics, I take the characteristics and partition based on each characteristic. And what do I do for test inputs? I take one partition I told you any set of input values representing that partition is good enough as a test input and each test input should belong to exactly one block from each characteristic.

And please remember when I choose characteristics and partition based on the characteristics, partitions must satisfy the property of being disjoint and complete.

(Refer Slide Time: 12:19)



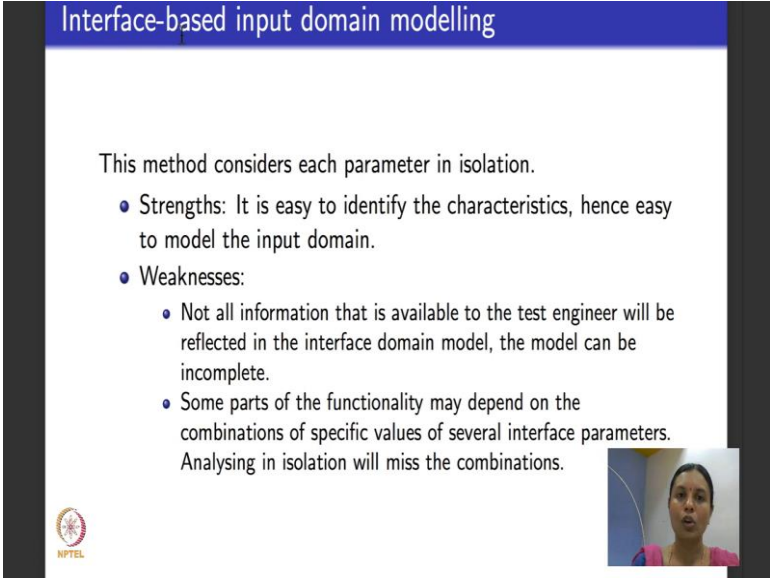
The slide is titled "Input domain modelling: Two approaches" in a blue header. It contains a bulleted list of points. In the bottom right corner, there is a small video inset showing a woman speaking. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.

- Input domain can be modelled in several different ways, needs extensive knowledge of the *domain*.
- Both valid and invalid inputs need to be considered.
- Two broad approaches available:
 - Interface-based approach
 - Functionality-based approach

So, what are the two approaches of input domain modeling? There are two popular approaches to do input domain modelling, one is called interface based approach and the other one is called functionality based approach. So, we will spend a few minutes understanding what each of these approaches are and two things to be observed, when I modelled the input domain I need to know the domain. What do I mean to mean by need to know the domain? I need to know what the program is going to do.

Let us say the program is a program that it is going to fly an aircraft I need to know a little bit about the application domain of the program. Let us say the program is programs that suppose to be in a banking sector then I need to know a little bit about the application domain. So, I need domain knowledge and my partition should consider both valid and invalid inputs because I am going to test the functionalities.

(Refer Slide Time: 13:11)



The slide is titled "Interface-based input domain modelling" in a blue header. The main content area is white and contains the following text and list:

This method considers each parameter in isolation.

- Strengths: It is easy to identify the characteristics, hence easy to model the input domain.
- Weaknesses:
 - Not all information that is available to the test engineer will be reflected in the interface domain model, the model can be incomplete.
 - Some parts of the functionality may depend on the combinations of specific values of several interface parameters. Analysing in isolation will miss the combinations.

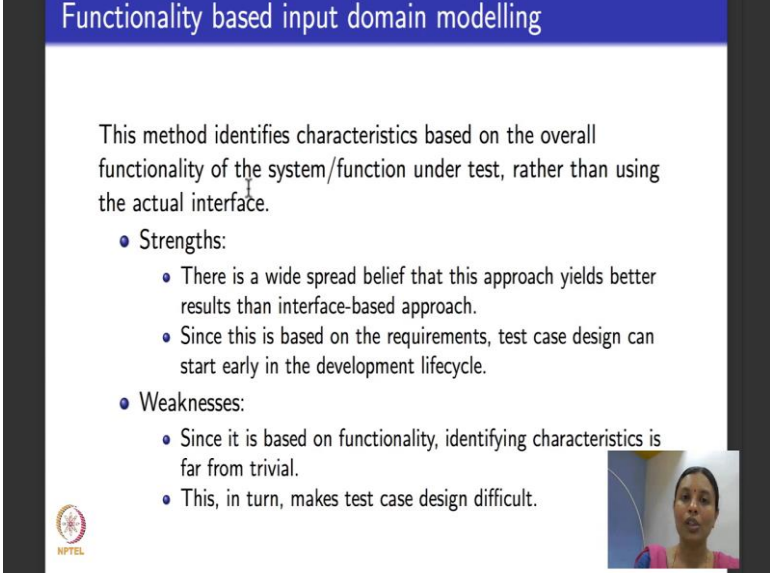
In the bottom left corner, there is a small circular logo with the text "NPTEL" below it. In the bottom right corner, there is a small video inset showing a woman with dark hair, wearing a pink top, speaking.

So, what we spend some time on now is, what are these two approaches to model input domain modelling? So, the first approach is what is called interface based input domain modelling. So, here what it does is that there are several characteristics of an input domain, they could be several parameters induced for each of these characteristics. Interface based approach considers each characteristic of parameter in isolation.

What are the possible strengths of interface based approach? Interface space approach is suppose to make it easier to identify the characteristics because we really do not worry about how do they influence each other, how one influences the other and what are the weaknesses? The weaknesses are, the problem is typically not all information is available to the test engineer that will be reflected correctly in the interface domain model when we consider each characteristic separately. So, there could be a risk of the input domain models that I consider. By models for input domain I mean sets, the sets that I consider for input domain could be incomplete and because I do not consider how they depend on each other, I might miss some functionality.

What I am trying to do in interface based approach is that I am trying to analyze each characteristic in isolation that might miss some functionalities that comes as a part of combination.



(Refer Slide Time: 14:32)



Functionality based input domain modelling

This method identifies characteristics based on the overall functionality of the system/function under test, rather than using the actual interface.

- **Strengths:**
 - There is a wide spread belief that this approach yields better results than interface-based approach.
 - Since this is based on the requirements, test case design can start early in the development lifecycle.
- **Weaknesses:**
 - Since it is based on functionality, identifying characteristics is far from trivial.
 - This, in turn, makes test case design difficult.

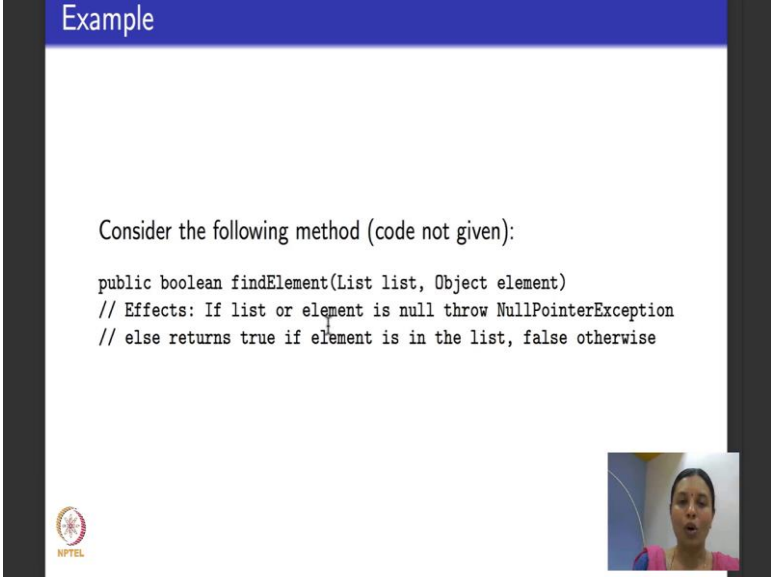
So, we will see examples that will help us to understand both the strengths and the weaknesses in detail. The next approach is functionality based input domain modelling. In this method characteristics are identified based on the overall functionality of a system. So, let us say I have a program under test or a function under test, I look at what that program or the function is supposed to do, what is its main functionality, why is it written, why does it exist, I look at those characteristics and then consider partitions of the input domain based on characteristics that represent the functionality of the considered program or function.

The strengths are that because I consider the main functionality this approach is believed to yield better results than interface based approach. In fact, there are empirical studies, I will give you some references about empirical studies in input domain partitioning, where this is validated and since it is directly based on functionality or requirements test case design can start very early in the development life cycle. Remember functionality caters to requirements and requirements are the first thing that are baselined in a typical development of software.

So, I test engineer can start designing test cases write the requirements face without waiting for the program to be ready. Of course, for execute in this test cases you need the program to be ready, but for designing the test cases I can start up front. What are the weaknesses? The weaknesses are because it is directly based on functionality you need domain knowledge, and identifying, partitioning the input domain modelled can be non-trivial.

So, this is a small example.

(Refer Slide Time: 16:01)



The slide is titled "Example" in a blue header. It contains the following text:

Consider the following method (code not given):

```
public boolean findElement(List list, Object element)
// Effects: If list or element is null throw NullPointerException
// else returns true if element is in the list, false otherwise
```

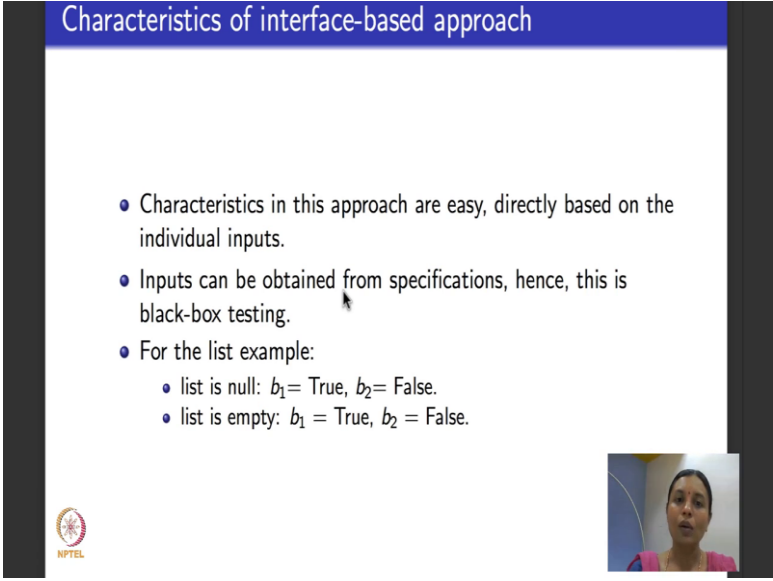
In the bottom left corner, there is a small NPTEL logo. In the bottom right corner, there is a small video inset showing a woman speaking.

This is an example of a code which is supposed to take a list as an input along with an element. If you remember we had looked at this code once before in one of the earlier weeks and it supposed to find out whether this element is present in the list or not. If the element is present in the list, it will return true that is why it suppose to return a Boolean value. If it is not present in the list it will return false. If the list itself or the element is null then will throw in exception null pointer exception.

Please remember I have not given you the code here. Why have we not given us the full code? We were not given the full code because as I told you in the last lecture we are, our focus is black box testing. In black box testing I do not design test cases based on the code. All the information that I need is which is the program you want me to test, this is the program of find element. What are it is inputs, it is inputs are list and the element to be found in the list. What is this program supposed to return?

It supposed to return a Boolean value true or false and when there are problems it suppose to return in null pointer exception. This is all the information that I need to know. I do not have to know the code for the program because I am doing black box testing. With this information can I use inputs base partitioning to design test cases for the program that is what we going to look at.

(Refer Slide Time: 17:22)



The slide is titled "Characteristics of interface-based approach" in a blue header. It contains three main bullet points. The first states that characteristics are easy and based on individual inputs. The second states that inputs are obtained from specifications, making it black-box testing. The third point, "For the list example:", has two sub-bullets: "list is null: $b_1 = \text{True}, b_2 = \text{False}$." and "list is empty: $b_1 = \text{True}, b_2 = \text{False}$." In the bottom right corner, there is a small video inset showing a woman speaking. The NPTEL logo is in the bottom left corner of the slide area.

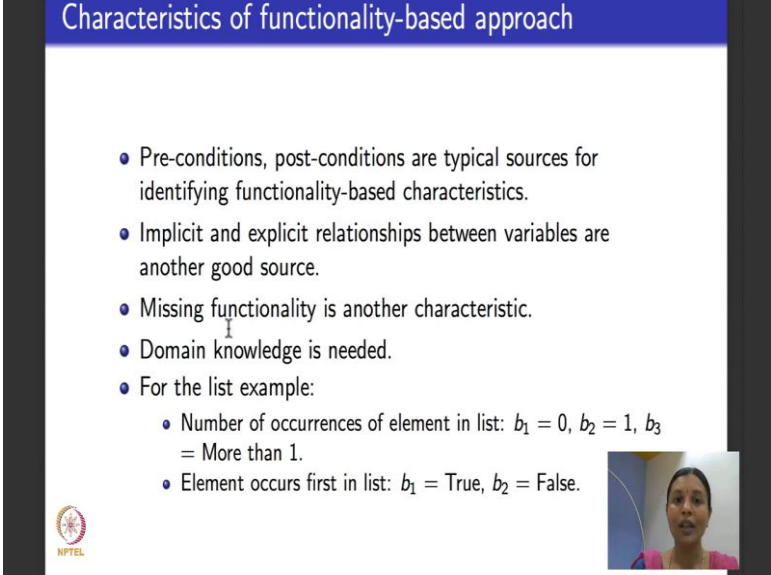
- Characteristics in this approach are easy, directly based on the individual inputs.
- Inputs can be obtained from specifications, hence, this is black-box testing.
- For the list example:
 - list is null: $b_1 = \text{True}, b_2 = \text{False}$.
 - list is empty: $b_1 = \text{True}, b_2 = \text{False}$.

Now, let us look at the characteristics for interface based approach for this program. What do I know? I know that this list, two inputs list and an element two possible outputs, true if the element is in the list, false if the element is not found in the list and an exception. So, simple interface based input domain partitioning will look like this. Is the list null, it will ask and the answer will be true if it is null, false if it is not null and then is the list empty, it will ask answer will be true, answer will be false. You could go ahead and write a few more things: is the list, is the element present in the list, but if when the moment when we talk about is the element present in the list, we are not considering interface based approach. Please remember that interface based approach considers each parameter, input parameter in isolation. For this program there are two inputs list and an element.

So, interface based approach directly considers the list and considers partitions of the kind of list only. It does not ask question about whether the second element input which the element, is it there in the list or not, does not ask that question. When we do

functionality based partitioning, we will consider that property into consideration that is what we do here.

(Refer Slide Time: 18:41)



The slide is titled "Characteristics of functionality-based approach" in a blue header. It contains a bulleted list of characteristics. The first four are: "Pre-conditions, post-conditions are typical sources for identifying functionality-based characteristics.", "Implicit and explicit relationships between variables are another good source.", "Missing functionality is another characteristic.", and "Domain knowledge is needed." The fifth bullet is "For the list example:", which has two sub-bullets. The first sub-bullet is "Number of occurrences of element in list: $b_1 = 0, b_2 = 1, b_3 = \text{More than 1.}$ ". The second sub-bullet is "Element occurs first in list: $b_1 = \text{True}, b_2 = \text{False.}$ ". In the bottom right corner of the slide, there is a small video inset showing a woman with dark hair wearing a pink top, speaking. The NPTEL logo is in the bottom left corner of the slide.

- Pre-conditions, post-conditions are typical sources for identifying functionality-based characteristics.
- Implicit and explicit relationships between variables are another good source.
- Missing functionality is another characteristic.
- Domain knowledge is needed.
- For the list example:
 - Number of occurrences of element in list: $b_1 = 0, b_2 = 1, b_3 = \text{More than 1.}$
 - Element occurs first in list: $b_1 = \text{True}, b_2 = \text{False.}$

So, when I do functionality based approach, for that list example, if you look at the bottom of this slide, we say is the element present in the list, how many times is it present in the list? The number of occurrences of element in the list, if it is 0 which means the element is not present in the list, if it is one then it is present in the list exactly once, if it is more than one then it is present in the list more than once. So, one characteristic based on which I derive the partitions is the number of occurrences of elements in the list: it does not occur which is the first one, it occurs exactly ones which is the second one, it occurs more than ones which is the third one.

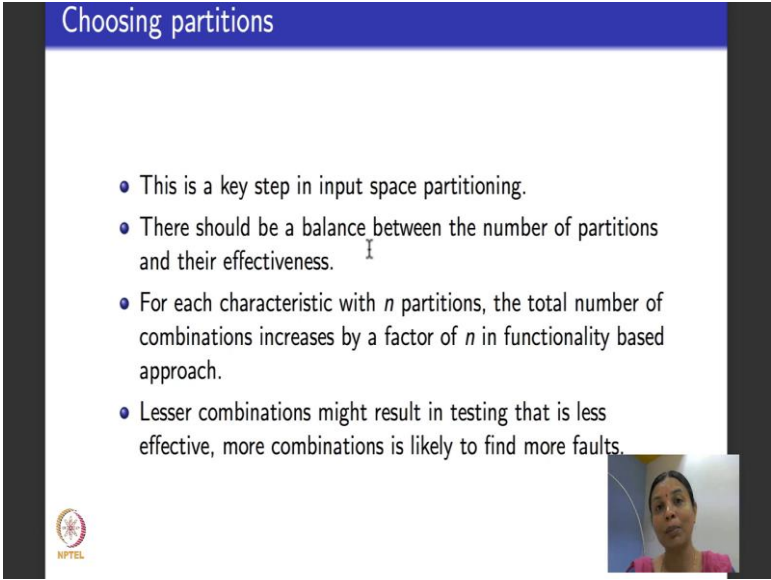
Now, if you see both the inputs to the program, the element and the list, are together considered in functionality based approach. When we did interface based approach, we considered only the list separately. So, another characteristic that we could arrive, this is just as an example for functionality based approach, is the following. We can ask does the element occur first in the list. Not a very interesting characteristic because the program is not worried about asking whether the element is occurring first in the list or not, but let us say I am interested in this characteristic for some weird reason. Then I can again write it I saying if it does occur first in the list, then one characteristic says true if it does not occur first in the list the next characteristics is false.

So, how do I do functionality based approach, where do I get these things from? Usually in a program preconditions and post conditions they are very rich sources to understand functionality. For example, if you look here right it is just a small little comment that acts as preconditions on what the program is expected to do.

So, I do not really need the code of the program. Just by looking at this I am able to arrive at characteristic right. The other things that people usually look, for testers usually look for, while writing characteristics, is to look at implicit and explicit relationships between variables. I will show you an example later in the lecture today about what this means. The other functionality people look for is typically what is missing functionality. So, that they can identify if the program is correctly handling missing functionality, and just to stress it again, to do all this unit extensive domain knowledge. Domain knowledge I mean the domain of this in which the software is running.

So, in the next step is choosing partitions. How do I choose the partitions? As I told you the number of partitions should not be too high. If you choose a very fine grain characteristic

(Refer Slide Time: 21:16)



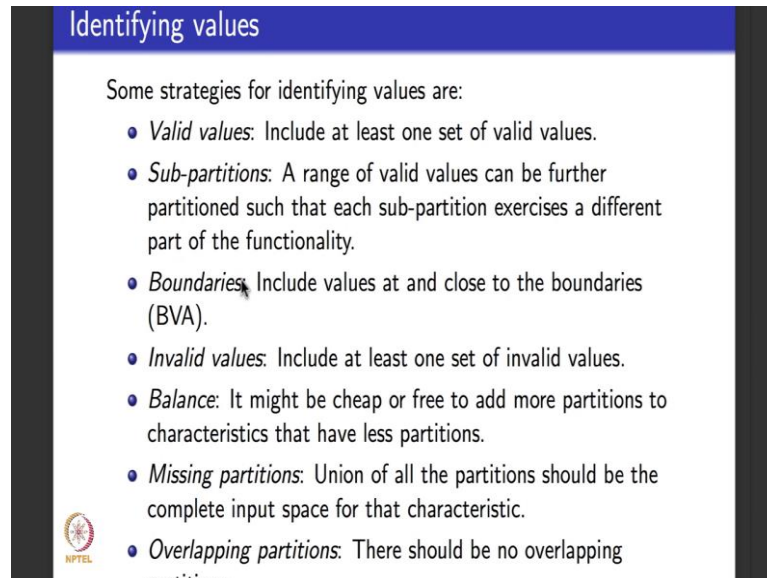
The slide is titled "Choosing partitions" in a blue header. It contains a bulleted list of four points. In the bottom right corner, there is a small video inset showing a woman speaking. The NPTEL logo is in the bottom left corner of the slide content area.

- This is a key step in input space partitioning.
- There should be a balance between the number of partitions and their effectiveness.
- For each characteristic with n partitions, the total number of combinations increases by a factor of n in functionality based approach.
- Lesser combinations might result in testing that is less effective, more combinations is likely to find more faults.

then you will end up in too many partitions and then if you choose one test input for each partition there will be too many test cases. Sometimes it might be needed to get clarity if error is very deep in the program to get focus on the error, but in general it involves an

experienced balance between choosing the partitions wisely such that they are not too large in number, but at the same time very effective in finding faults.

(Refer Slide Time: 21:42)



The slide is titled "Identifying values" in a blue header. Below the header, it says "Some strategies for identifying values are:" followed by a bulleted list of seven strategies. The strategies are: Valid values, Sub-partitions, Boundaries, Invalid values, Balance, Missing partitions, and Overlapping partitions. Each strategy is described in a sentence. There is a small NPTEL logo in the bottom left corner of the slide content area.

Identifying values

Some strategies for identifying values are:

- *Valid values*: Include at least one set of valid values.
- *Sub-partitions*: A range of valid values can be further partitioned such that each sub-partition exercises a different part of the functionality.
- *Boundaries*: Include values at and close to the boundaries (BVA).
- *Invalid values*: Include at least one set of invalid values.
- *Balance*: It might be cheap or free to add more partitions to characteristics that have less partitions.
- *Missing partitions*: Union of all the partitions should be the complete input space for that characteristic.
- *Overlapping partitions*: There should be no overlapping partitions.

How do I identify values within partitions? Typically partition should be such that all these things are used to find the test input values. I should find a few valid values, means there should be in that list example, there should be a nice long list and a case of element being present in the list at least once, and sometimes I might have to consider one partition in isolation and partition it further. There could be needs for that. Sometimes we have to identify values exactly at boundary.

If you remember in the last lecture I told you that once I do equivalence class partitioning I can also do boundary value analysis that is what this is about, and the some partitions have to give me invalid values, because I need to know whether the program has exception handling features for all these things and as I told you there must be a balance between the number of partitions and effectiveness in finding test cases. And there should not be any missing partitions, I should get back the entire input domain that is very important, it should be complete.

And as I told you there should not be any overlapping partitions, the partitions should be disjoint.

(Refer Slide Time: 22:49)

TriTyp Example

- We re-visit the program TriTyp that determines the type of a triangle, given three sides as input.
- Interface-based partitions of inputs:

Partition	b_1	b_2	b_3
q_1 ="Relation of Side 1 to 0"	> 0	$= 0$	< 0
q_2 ="Relation of Side 2 to 0"	> 0	$= 0$	< 0
q_3 ="Relation of Side 3 to 0"	> 0	$= 0$	< 0
- Consider the partition q_1 for Side 1. If one value is chosen from each partition, the result is three tests. Choose test inputs as Side1=7 in the first one, Side1=0 in the second test and Side1=-3 in the third test.
- Some partitions represent valid values, some represent invalid values.

So, what will do for the rest of today's lecture is we will take one example and we will go through what are the various ways in which we can partition the input domain for that example. So the example that I have chosen is the example that we did when we did logic coverage criteria, because that was the most recent example that we visited. So, I thought it will be fresh in your mind. So, it is good to revisit that example. So, the example that we did was the type of a triangle example abbreviated as TriTyp. If you remember we had this program in the last week where we looked at this is TriTyp program which took 3 sides of a triangle as input and determined what was the kind of triangle. Was it in equilateral triangle, isosceles triangle or was it an invalid triangle and so on.

So, here is an example of how an interface based partition for that will look like. For example, I could say that there are 3 sides: side 1, side 2, side 3 and I could consider the relationship of the 3 sides to be with 0. So, I could say what is side one, is side one greater than 0 a side one and number length of side one a number equal to 0, is length of side one a number less than 0 that could be one partition. The next partition could be the same thing for side two is the length of side two a number greater than 0, equal to 0, less than 0. The next partition could be related to the same thing for side 3 is the length of side 3 greater than 0, equal to 0 or less than 0.

So, suppose I consider partition q one for side one what do I do? I choose one value from each partition which will result in 3 tests right. One for side one greater than 0 like for example, you could choose 7 or any number greater than 0 as the value. The second one says side one is equal to 0 that is what I have chosen as a second value. Third one choose a negative number side one less than 0. So, if you see here what do these two represent side 1 equal to 0, side 1 minus 3 which is less than 0, they represent 0 invalid values. So, what do they test about the program? They test whether the program handles these invalid triangles correctly. So that also satisfies this condition that I told you here that some partitions should include few invalid values.

So, this is one partition which checks for greater than 0, equal to 0, less than 0.


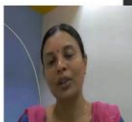
(Refer Slide Time: 25:05)

TriTyp Example, contd.

Another possible interface-based partitioning for TriTyp:

Partition	b_1	b_2	b_3	b_4
q_1 ="Relation of Side 1 to 0"	> 0	$= 0$	$= 1$	< 0
q_2 ="Relation of Side 2 to 0"	> 0	$= 0$	$= 1$	< 0
q_3 ="Relation of Side 3 to 0"	> 0	$= 0$	$= 1$	< 0

- This allows us to consider comparison to 1 too.
- These partitions are complete as the sides are integers. They will not be complete if the sides are floating point numbers.

Sometimes you might want to refine it into doing 4 partitions. You might want to do greater than 0, equal to 0, separately maybe equal to one and less than 0. This is just a choice, just to give you an illustration of an alternative option to do partition. In this case what will happen? There will be 4 test inputs: one for a side one greater than 0, 1 for side one equal to 0, 1 for side one equal to 1 and one for side one being a negative number. Similarly for side 2, similarly for side 3.

Now, another subtle point that I would like to like you all to understand and observe here is that if you remember the program TriTyp in input to the program were 3 integers. If you remember all the 3 sides sorry input to the program were 3 sides and all the 3 sides

what was their data type? They were all integers, these partitions are valid provided the 3 sides are integers. Suppose I will make the 3 sides of the triangle is floating point numbers, then this way of doing partitioning is not valid. Why is it not valid, because it will not be complete. If you see these 4 partitions, they, if they are floating point number then between these 4 partitions none of the partitions test for this length of a side of a triangle being a number between 0 and 1.

Let us say the length of a side of a triangle is 0.7, it does not test for that, it does not belong to any of these partitions. So, it does not satisfy the completeness criteria, but for that program that we had our type of the 3 sides was integers. When I consider integers these partitions are valid.



(Refer Slide Time: 26:54)

TriTyp Example, contd.

Functionality-based partitioning for TriTyp:
This partitioning is based on the type of the triangle, which is the main functionality of the TriTyp program.

Partition	b_1	b_2	b_3	b_4
$q_1 = \text{"Geometric classification"}$	scalene	isoceses	equilateral	invalid

Partitions above are *not* disjoint.

So, you have to be very careful in determining whether partition is valid or not. Based on the type of the domain, one partition could be valid for one domain type and could be invalid for another two domain type. The same partition could be invalid for another domain type. So, when I do functionality based partitioning of TriTyp, I do not directly look at inputs in isolation I start looking at the functionality of the program. What was the program TriTyp supposed to do? It was supposed to determine the type of a triangle. What were the 4 types of outputs that it was suppose to produce? It was supposed to produce numbers as 0, 1, 2, 3 to tell you what the kind of triangle it was. It was supposed to tell you whether the triangle was scalene or isosceles or equilateral or invalid.

So, functionality based approach will consider the functionality of a program, what is the main functionality of the program and try to come up with the characteristic based on the functionality. In the case of type of a triangle the functionality was the type of the triangle itself. So, here is one partitioning. So, I say my criteria for partitioning is the classification for the type of a triangle and then there are 4 partitions: triangle the 3 sides are such that the triangle is scalene, the 3 sides are such that the triangle is isosceles, the 3 sides are such that the triangle is equilateral, and one of the sides or maybe more than one of the sides are such that the triangle is not valid triangle.

So, suppose I consider a partitioning this way. Before we move on in design test cases as I told you the first thing to understand is our partitions complete our partitions disjoint you will realize that these partitions are not disjoint. If you think for a minute can you tell me why these partitions are not disjoint, what is the problem? The problem is if you see every equilateral triangle is also an isosceles triangle. What is an equilateral triangle, it means all the 3 sides are equal. What is isosceles triangle, it says any two sides are equal. So, when all the 3 sides are equal any two sides are equal. So, these partitions suppose I give a triangle that is equilateral it will belong to b₃, partition and it will belong to b₂ partitions. So, it does not satisfy the property of disjointness.



So, how do I overcome this small issue? The tweak or the correction that I have to do for the partitions is very simple.

(Refer Slide Time: 28:53)

TriTyp Example, contd.

Functionality-based partitioning for TriTyp that is disjoint.

Partition	b_1	b_2	b_3	b_4
q_1 ="Geometric classification"	scalene	isoceses, not equilateral	equilateral	invalid



So, I just rewrite the partitions to mean that it is scalene, it is a isosceles, but not equilateral, it is equilateral and it is an invalid triangle. What will this achieve this will make sure that here for b_2 , I do get a triangle that is only isosceles. It will separate equilateral triangles that also have the scope for being isosceles. So, it says isosceles triangle not equilateral. This way I made sure that the partitions b_2 and b_3 which were overlapping here become disjoint here.

(Refer Slide Time: 29:30)

TriTyp Example, contd.

Possible test inputs for functionality-based partitioning for TriTyp that is disjoint.

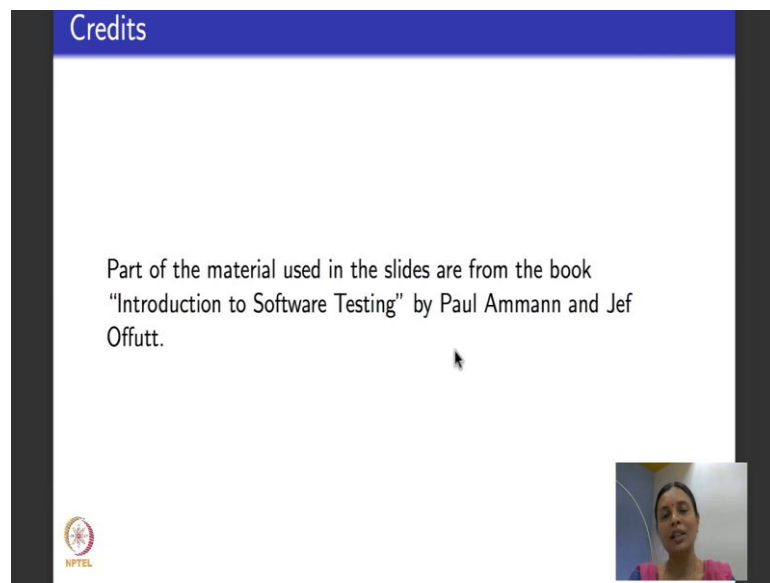
Param	b_1	b_2	b_3	b_4
Triangle	(4,5,6)	(3,3,4)	(3,3,3)	(3,4,8)

Now, I am ready to give test inputs to get to start testing the program. The test inputs that I could give are one set of test cases that will test with other triangle is scalene and for example, 4, 5, 6. This is good enough to test for triangle scalene, it will be this is another set of test cases with two sides being equal to test if a triangle is isosceles, but not equilateral. The third set of test cases with all 3 sides equal have given 3, 3, 3, give any values all of them need to be the same, 4, 4, 4, 5, 5, 5, anything is alright. This tests for equilateral triangles this tests for not a valid triangle is this clear.

So, if you see throughout what we have done we looked purely at the input looked at what input values could be given and how they could be partitioned, based on just the input values which is interface based approach, which was discussed here or based on the overall functionality, which is the functionality based approach which was discussed here.

(Refer Slide Time: 30:28)



So, to design test cases for input domain partitioning, I do not really need the program code. I need knowledge about the inputs and I need to know what the functionality of the program is and I could partition the input domain in several different ways. What will do in the next lecture is, we look at once I have come up with the partitions, how do I design test cases? What are the various coverage criteria to design test cases? Do I pick one value from each partition, do I pick one value from each partition such that the every other value from other partition is covered, do I pick two values from each partition, is there one partition that I have to focus on? All these things we will understand in the next lecture.

Thank you.

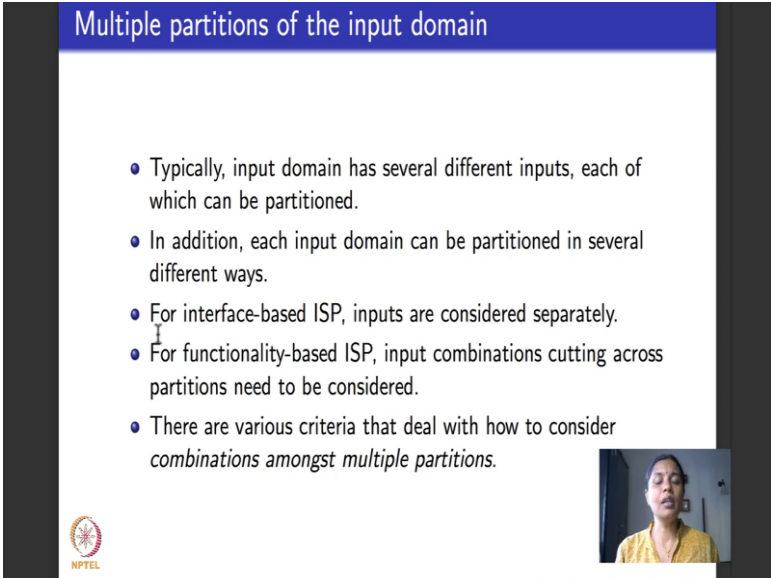
Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 33
Input Space Partitioning: Coverage Criteria

Hello again. Welcome to week 7, this is a fourth lecture. I will try and finish test case generation based on input space partitioning.



What did we look at till this week? We saw functional testing; I told you what are the various kinds of functional testing. Then I told you what is the basics of input space partitioning. In the last lecture we defined partitions. I told you how the input space could be partitioned in several different ways. One way we looked at was what was called interface domain part based partitioning; the other one was functionality based partitioning. So, we revisited the triangle type example. And then we saw how each of these work.

(Refer Slide Time: 00:52)



Multiple partitions of the input domain

- Typically, input domain has several different inputs, each of which can be partitioned.
- In addition, each input domain can be partitioned in several different ways.
- For interface-based ISP, inputs are considered separately.
- For functionality-based ISP, input combinations cutting across partitions need to be considered.
- There are various criteria that deal with how to consider *combinations amongst multiple partitions*.

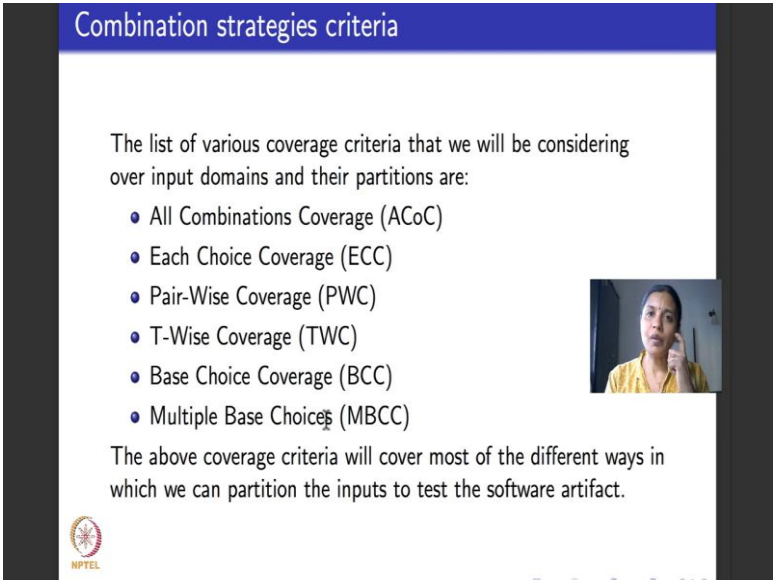
 

Today what will be seeing is what are the various coverage criteria that we can define based on the partitions. Typically when I consider input domain there is one domain for every kind of variable that is a part of the input. And there are several different inputs each have their own types, each have their own domain and each of these input domains

can be portioned in several different ways based on what is the requirement that I am considering and what is the requirement that I am trying verify.

For interface based input space partitioning we consider input separately as we saw in the last lecture. For functionality based input space partitioning we partition the input based on the concerned functionality of the program under test. So, now, what we will see is that how do we combine all these various partitions of inputs? Is there a clever way to combine, a better way to combine, an easier way to combine or a focused way to combine these so that we can test them.

(Refer Slide Time: 01:50)



The slide is titled "Combination strategies criteria" in a blue header. The main content area is white and contains the following text and list:

The list of various coverage criteria that we will be considering over input domains and their partitions are:

- All Combinations Coverage (ACoC)
- Each Choice Coverage (ECC)
- Pair-Wise Coverage (PWC)
- T-Wise Coverage (TWC)
- Base Choice Coverage (BCC)
- Multiple Base Choices (MBCC)

The above coverage criteria will cover most of the different ways in which we can partition the inputs to test the software artifact.

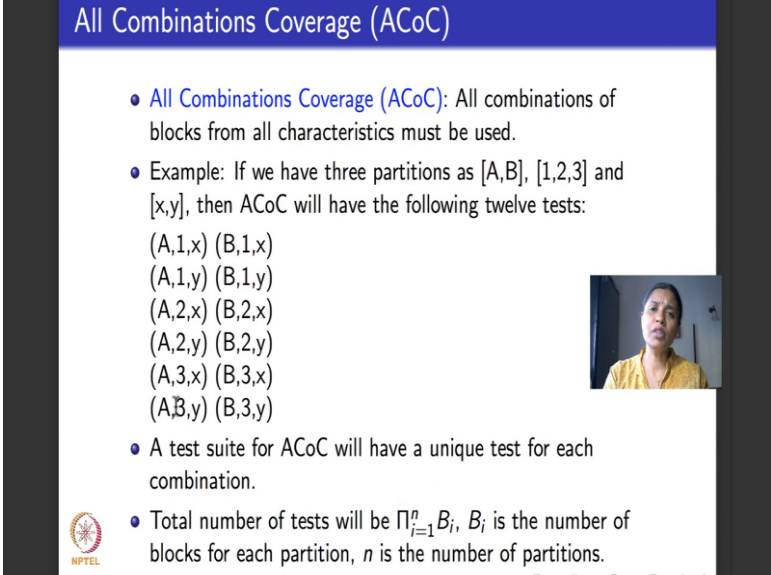
In the bottom right corner of the slide, there is a small video inset showing a woman with dark hair wearing a yellow top, speaking. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.

So, what we will be seeing today a various coverage criteria again define independent of any examples. I will explain these coverage criteria using an abstract example. In the next lecture we will take a concrete example which will be the triangle type program that we saw and see how these various coverage criteria can be used to define functional test cases for the triangle type program. So, the various coverage criteria that we will be defining in this lecture I listed here.

We will work with six coverage criteria. First one will be all combinations, second one will be each choice, third one and fourth one will take pairs or will take tuples of length t , the fourth and the fifth one. I mean the fifth and the sixth one will fix one coverage criteria like a base choice, and then define criteria based on that. Between these criteria


almost all that pa ways of partitioning and input that we know off till date in test case generation will be covered.


(Refer Slide Time: 02:54)



All Combinations Coverage (ACoC)

- **All Combinations Coverage (ACoC):** All combinations of blocks from all characteristics must be used.
- **Example:** If we have three partitions as [A,B], [1,2,3] and [x,y], then ACoC will have the following twelve tests:
(A,1,x) (B,1,x)
(A,1,y) (B,1,y)
(A,2,x) (B,2,x)
(A,2,y) (B,2,y)
(A,3,x) (B,3,x)
(A,3,y) (B,3,y)
- A test suite for ACoC will have a unique test for each combination.
- Total number of tests will be $\prod_{i=1}^n B_i$, B_i is the number of blocks for each partition, n is the number of partitions.





So, we begin with all combinations coverage, abbreviated as ACoC. What does it do? It says you take input domain you partition the input domain the of the various inputs, you consider every combination from every partition. So, it is just this exhaustive testing based on partitions of the input domain. So, all combinations of all the blocks of partitions with respect to all the characteristics must be used to be able to test it for ACoC.

Suppose, let us take for some piece of software, for some input we have the following 3 partitions. Let us not worry about what the input is just an abstract representation of the 3 partitions. One partition partitions into 2 categories into sets called A and B, another input is partitioned into 3 sets call it 1, 2, 3, the third input is partitioned into 2 sets call it x and y right. So, there are 3 inputs: first input is partition in 2 ways A and B, second input is partitioned in 3 ways 1, 2 and 3, third input is partitioned in 2 ways again, x and y. What does all combinations say? It says take every partition with every other partition to test.

So, what I do for the first input I choose partition A and for the first input I choose partition B here, then I keep the partition for the second and the third fixed to be one and x, 1 and x. So, here I vary the partitions for the first input from A to B, similarly for the

second input, I consider partition one, I fix partitions 1 and y for the second and third input I again vary the partition one for the first input from A to B. Similarly, what I do here? Now I vary the partitions for the second input I have made it changed it from one I have made it 2 here I again have made it two. So, I consider A and B with partition 2 and x.

So, between the middle column I cover the partitions of the second input, in the third column I cover the partitions of the third input, in the first tuples here that I am running my cursor down I cover the partitions of the first input. So, the first input has 2 partitions A and B, I have varied them thoroughly, second input has 3 partitions 1, 2 and 3 the second tuple in each of these test cases cover that, the third input domain has 2 partitions x and y, the third input in each of these tuples cover that.

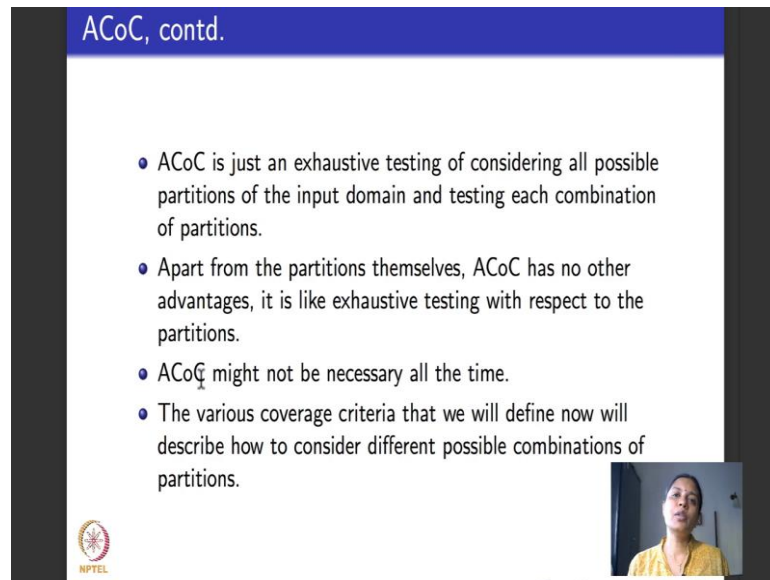
So, to be able to, this is a test requirement for all combinations coverage criteria. To be able to actually test it, what do I have to do? I have to pick up one test case for each of these petitions. So, I have to pick up one test case where the value is from, value for the first input is from A, here one test case whether value of the second input is from one and one test case whether value of the third input is from the partition x. Similarly for the second TR, I have to pick up one test case whether value of the first input is from partition B, the value of the second input if from partition one and value of the third input is from partition x and so on.

So, now I you see how many different partitions can be there for all combinations coverage criteria? Let us say each input is partitioned in B_i ways, each input is partitioned in B_i different ways and then there are n inputs, then the total number of partitions that I am targeting in all combinations coverage criteria will be a product of all these values B_i . This \prod that is written here means it is a product, it is a product

$\prod_{i=1}^n B_i$ where B_i is the number of blocks that are there for each partition.

So, if you go back to this example, the first input has 2 partitions number is 2, B_i is 2 for that, second input the B_i which is b_2 is 3 for that, for the third input the third partition B_3 is 2. So, the total number of test cases will be 2 into 3 into 2 that will be 12 and if you see 12 test cases have been written here.

(Refer Slide Time: 07:03)



ACoC, contd.

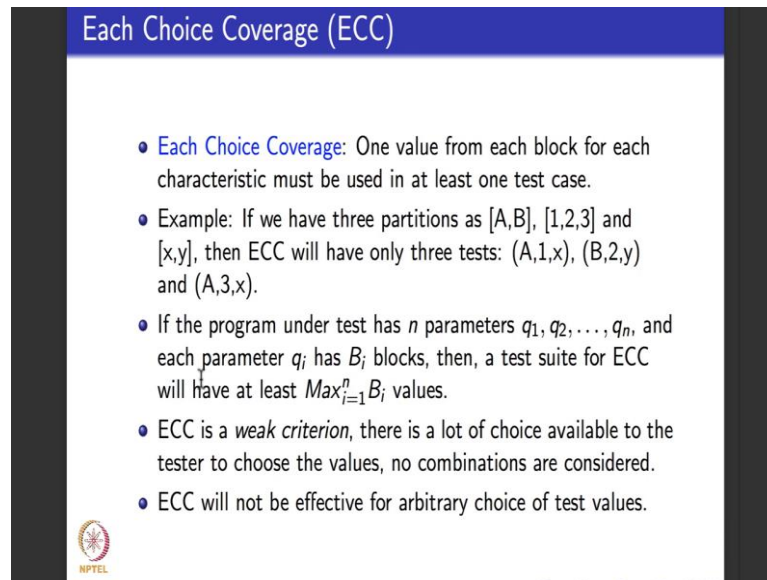
- ACoC is just an exhaustive testing of considering all possible partitions of the input domain and testing each combination of partitions.
- Apart from the partitions themselves, ACoC has no other advantages, it is like exhaustive testing with respect to the partitions.
- ACoC might not be necessary all the time.
- The various coverage criteria that we will define now will describe how to consider different possible combinations of partitions.

The slide features a blue header with the title 'ACoC, contd.' and a white body with a bulleted list. In the bottom right corner, there is a small video inset showing a woman speaking. The NPTEL logo is visible in the bottom left corner of the slide area.

Now, what is ACoC? ACoC or All Combinations coverage Criteria is not intelligent at all, it just tells you generate the partitions and just do an exhaustive testing with reference to all partitions. It is somewhat like doing all combinations coverage that we differ logic coverage criteria if you remember. Obviously, if we generating the partitions to be able to test them in a slightly intelligent way, we not generating the partitions to be able to test them before all possible combinations.


So, ACoC is not considered a great test requirement coverage criteria and it really has no advantages. We begin with it because we define, we would like to understand it for completeness sake and it is one of the value will coverage criteria. Whenever the partitions are small maybe we could afford to do ACoC coverage criteria. And the thing is it may not be necessary all the time. Now, how do we prevent ourselves from testing all the possible combinations? Are there better choices?

(Refer Slide Time: 08:01)



Each Choice Coverage (ECC)

- **Each Choice Coverage:** One value from each block for each characteristic must be used in at least one test case.
- **Example:** If we have three partitions as [A,B], [1,2,3] and [x,y], then ECC will have only three tests: (A,1,x), (B,2,y) and (A,3,x).
- If the program under test has n parameters q_1, q_2, \dots, q_n , and each parameter q_i has B_i blocks, then, a test suite for ECC will have at least $\text{Max}_{i=1}^n B_i$ values.
- ECC is a *weak criterion*, there is a lot of choice available to the tester to choose the values, no combinations are considered.
- ECC will not be effective for arbitrary choice of test values.



Now, the next coverage criteria that we will see is what is called each choice coverage abbreviated as ECC. ACoC says you take all combinations of all the partitions, each choice coverage is the other extreme of ACoC. It says from each partition you pick up only one value. So, what is each choice coverage say, it says you pick up one value from each block, block is the same as partition, you pick a one value from each block for each characteristic and use it as your TR.

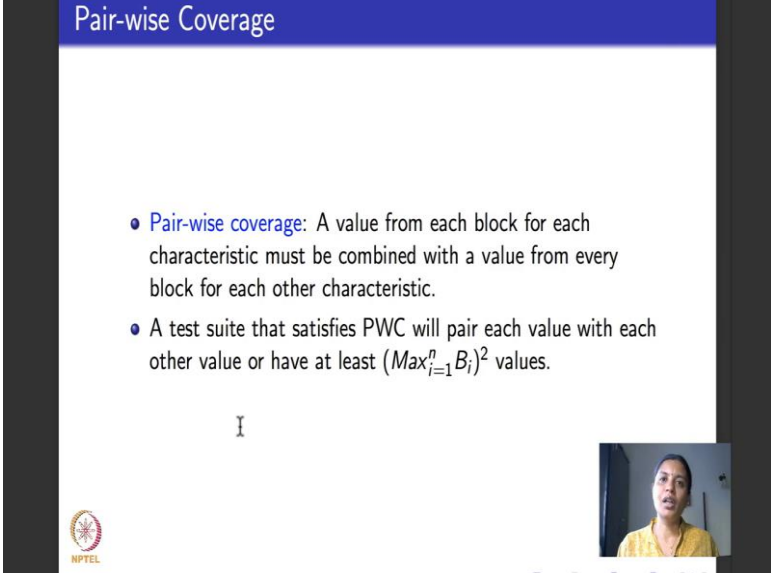
So, for the same example that we had that the first one was partition was A B, second one was 1, 2, 3, third one was x, y each choice coverage will have only 3 test cases. Randomly pick up, let us say I have picked up A for the from the first partition, I picked up 1 from the for second partition, and I picked up x from the third partition. In this case I pick up B from the first partition, 2 from the second partition let us say Y from the third partition.

Now, I have more or less covered the 2 partitions A and B and x and y. The only thing that I have not yet covered is this partition 3 for partitioning the second input. So, I pick up that for the third case and I can substitute either A or B for the first one, or x or y for the second one. So, I have chosen A, 3, x. So, each choice coverage says pick one value from each block. That is all we have done here. So, is the program under test has n parameters let say q_1 to q_n , and each parameter has let say B_i blocks or B_i partitions then what will the test suite for ECC will have? It will have at least max of B_i values.

Why is max? If you as you can clearly see here in the 3 partitions that we had for this example, only the second one 1, 2, 3 had cardinality three. So, to cater to that each choice thing we have to pick up one once 2 once and 3 once. So, and that is the maximum number. So, we need maximum of those test case values.

Like all combinations coverage criteria was like not a clever one, it included everything. ECC test the other approach it just says pick up anything randomly. So, it is actually a very weak coverage criteria. There is a lot of choice available on what you can pick up and what you need not and you might leave out some important combinations while testing with ECC. So, ECC will not be effective for arbitrary choice of test values.

(Refer Slide Time: 10:33)



The slide is titled "Pair-wise Coverage" in a blue header. It contains two bullet points:

- **Pair-wise coverage:** A value from each block for each characteristic must be combined with a value from every block for each other characteristic.
- A test suite that satisfies PWC will pair each value with each other value or have at least $(\text{Max}_{i=1}^n B_i)^2$ values.

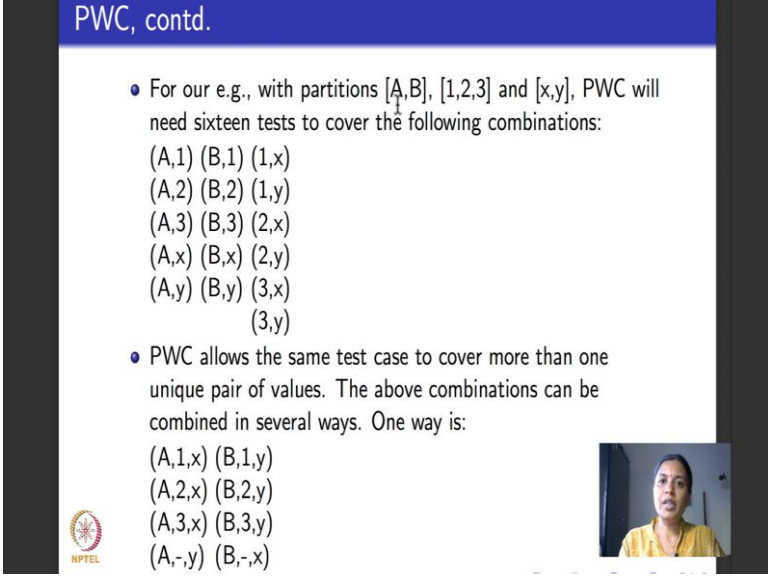
Below the text is a small, faint diagram of a vertical bar with the letter 'I' next to it. In the bottom right corner, there is a small video inset showing a woman in a yellow shirt speaking. The NPTEL logo is in the bottom left corner.

So, what do people do? People look for midway options between ECC and all combinations coverage criteria. One midway option is what is called pairwise coverage. What happens in pairwise coverage? In pairwise coverage a value from each block of partition for each characteristic is combined with a value from every other block or partition for the other characteristic. So I take one pa block or partition corresponds to one characteristic of an input domain, I pick this value

Now, I list out what an all what are the other partitions that I have left out at. I park this value, fix this value and then pairwise combine it with all other values. So, a test suite that satisfies pairwise coverage, will have will pair each value with each other value or it will have how many test cases if B_i are there, B_i is the cardinality of each block or each

partition, then it is clear that it will have max i is equal to 1 to n and B^2_i values right because I fix one B_i and then I will let it vary over all the other partitions, then I fix the next one I will let it vary over all the partitions. So, it is B_1 into this B_2 into this and goes on. So, it will be max of B_i , the whole squared.

(Refer Slide Time: 11:45)



PWC, contd.

- For our e.g., with partitions $[A, B]$, $[1, 2, 3]$ and $[x, y]$, PWC will need sixteen tests to cover the following combinations:
 - (A,1) (B,1) (1,x)
 - (A,2) (B,2) (1,y)
 - (A,3) (B,3) (2,x)
 - (A,x) (B,x) (2,y)
 - (A,y) (B,y) (3,x)
 - (3,y)
- PWC allows the same test case to cover more than one unique pair of values. The above combinations can be combined in several ways. One way is:
 - (A,1,x) (B,1,y)
 - (A,2,x) (B,2,y)
 - (A,3,x) (B,3,y)
 - (A,-,y) (B,-,x)

So, for the same example we had this A,B, 1, 2, 3 and x, y, 3 different ways of partitioning pairwise coverage. How many test will I need? I will need 16 different tests. Why so? It the way they are arranged it will be clear to you what I have done here. I have fixed a to be the thing that I want to consider in the first partition, I let the second tuple vary I vary to over the second partition 1, 2 and 3, I vary it over the third partition x, y. In this column here I have fixed b is from the first partition, I vary it over the tuple 1, 2 and 3 and I have vary it over the tuple x, y for the third partition.

Now, between these I have not covered the way one varies with reference to the third partition x, y, the way 2 varies with reference to the third partition x, y, the way 3 varies with reference to the third partition x, y. That is what I have done here in the third column. In the third column you take combinations with reference to the partitions 1, 2, 3 and x, y and choose pairs that let each of them vary. So, here I fix one let x and y vary, here I fix 2, let x and y vary, here I fix 3 let x and y vary again, right.

So, if you count how many different test cases will be there, there will be so many different test cases. But remember one thing we are doing too much of work here. In fact,

pairwise coverage what is let us do? It allows the same test case to cover more than one unique pair of values in some sense the above combinations that we have listed out here they can be combined in several ways. For example, I could directly do A, 1, x, in which case I have done paired A with 1, and 1 with x. If I do A, 2, x then I have paired A with 2 and I have paired 2 with x.


So, in the first case I do A with 1, 1 with x, I get A, 1, x, then I do A with 2, 2 with x, I get A, 2, x. Similarly A, 3, x considers both the pairs: a pairing of A with 3, and A pairing of 3 with x. So, once I have done here I just need to pair A with y, and because I have already paired A with 1, 2 and 3 once I have put a dash here in the last line, when I pair A with y. Read that dash as I can feel free to choose any of 1, 2 or 3 for that, because I have already paired A with each of them when I pair A with y, I can choose any one of 1, 2 or 3.

Similarly, in the other case what I have done here? We have paired B with 1 and y together, B with 2 and y, B with 3 and y. What is left to be paired? B and x; while pairing B and x because b is already paired with 1, 2 and 3, I put a dash to indicate that I can choose any of 1, 2 or 3 while pairing B, is that clear.

(Refer Slide Time: 14:43)

T-wise Coverage

- A natural extension of PWC is to require t values instead of pairs.
- **T-Wise Coverage:** A value from each block for each group of t characteristics must be combined.
- If the value for T is chosen to be the number of partitions, then TWC is the same as ACoC.
- A test suite that satisfies TWC will have at least $(\max_{i=1}^n B_i)^t$ values.
- TWC is expensive in terms of the number of tests, empirical studies indicate that going beyond PWC is mostly not useful.


I

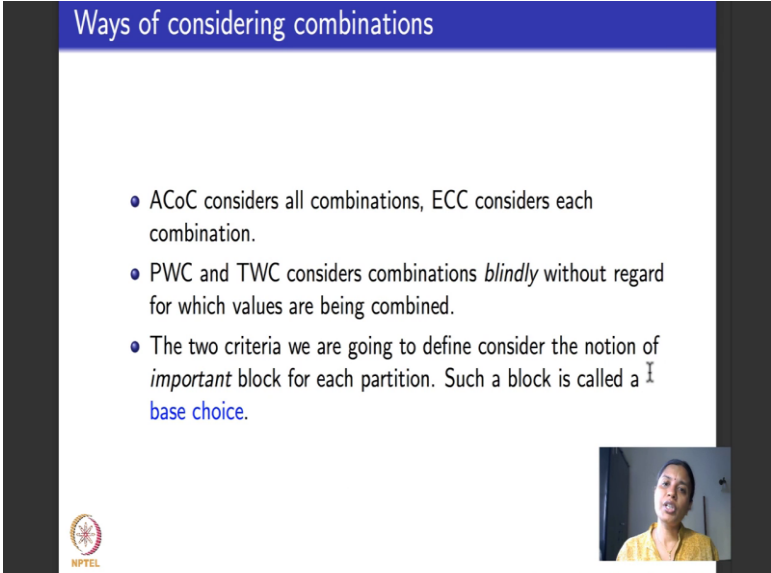
So, moving on pairwise considers only pairs or sets of 2. Whenever we to sets of 2 usually we can also extended to a arbitrarily large number. So, pairwise coverage can be extended to T wise coverage. So, instead of looking at a pair of values we require T

values. So, what is T wise coverage? It says a value from each block or partition for each group of T characteristics must be combined. If the value for T is chosen to be the number of partitions that is the entire number, then T becomes all combinations coverage criteria.

If T is chosen to be one then T becomes each choice criteria. If T is chosen to be 2 then T becomes pairwise coverage criteria. A test suite that satisfies T wise coverage criteria as an extension of pairwise coverage criteria will have $\max_i B_i \leq n$, where B_i is the power of T values where B_i is the same, it is a cardinality for each of the partitions.

Again like all combinations coverage criteria T wise coverage criteria is considered to be an expensive coverage criteria, and is usually considered to be almost close to exhaustive testing or not necessary at all. So, people usually say that it is not wise to go beyond pairwise coverage criteria. An empirical study in software testing that involve input space partitioning also prove that T wise coverage criteria is not a very useful coverage criteria. But again we define it for the sake of completeness to indicate that whenever it is necessary it is possible to choose the T of your own. T could be 3, 4, 5 if we have lots of partitions and then do T wise coverage criteria.

(Refer Slide Time: 16:26)



The slide is titled "Ways of considering combinations" in a blue header. It contains three bullet points:

- ACoC considers all combinations, ECC considers each combination.
- PWC and TWC considers combinations *blindly* without regard for which values are being combined.
- The two criteria we are going to define consider the notion of *important* block for each partition. Such a block is called a *base choice*.

In the bottom right corner, there is a small video inset showing a person speaking. In the bottom left corner, there is the NPTEL logo.

So, now what are the coverage criteria is that we have seen so far? We have seen four different coverage criteria. We began with all combinations coverage criteria which was like exhaustive testing with reference to combinations then we went on to the extreme

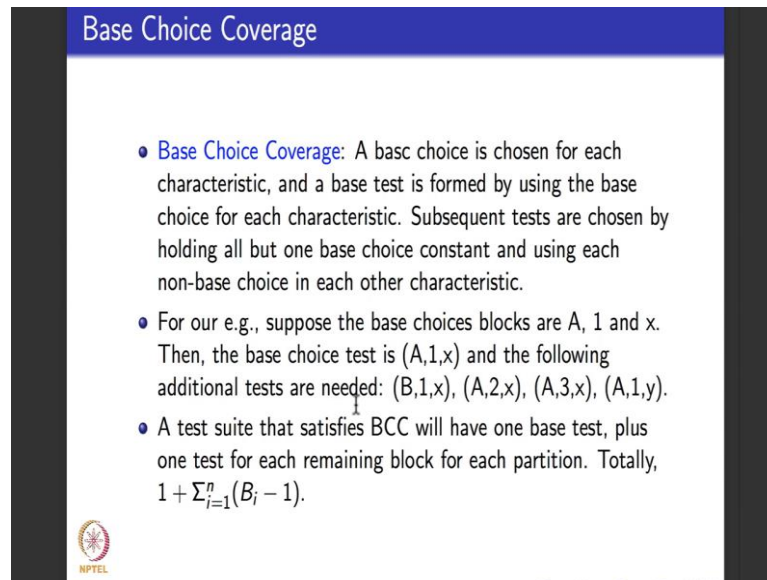
down side. We said we will take each choice coverage criteria, pick up one choice from every partition, and then we did midway which was pairwise or T wise where the value of T can be chosen by the user.

Now, all these coverage criteria put together have a bit of a disadvantage. What is the disadvantage that they have? They consider the combinations blindly. What do I mean by blindly? They do not really think what is to be useful what categories of combinations would yield faults with a higher chance. Maybe you would want to focus on a particular set of input values. Can I focus and look at only those input values and consider the partitions with reference to the others? They do not look at all these they just look at things blindly.

Now, we look at 2 more coverage criteria that focus on avoiding these blind combinations of partitions. Those 2 coverage criteria will depend on fixing one partition as a base choice. So, the base choice is what we called as an important partition or an important block for a partition of a particular input and we define coverage criteria on that base choice. Now if you go back to our lessons on logic coverage criteria, base choice coverage counterpart in logic coverage criteria would be what is called active clause coverage. If you remember when we did logic coverage criteria, we said I want to focus on one particular clause and see how that clause influences the predicate and the clause that I want to focus on will be called active clause. Similarly we had inactive clause and we defined coverage criteria that will let each clause in turn to be reactive clause and test right.


Similarly, here when we look at partitions, they might be combinations or characteristics of partitions that I want to focus on to see how it influences particular pieces software under test. So, that block or partition that I want to focus on is called base choice coverage and we look at 2 different coverage criteria based on the base choice coverage.

(Refer Slide Time: 18:48)



Base Choice Coverage

- **Base Choice Coverage:** A base choice is chosen for each characteristic, and a base test is formed by using the base choice for each characteristic. Subsequent tests are chosen by holding all but one base choice constant and using each non-base choice in each other characteristic.
- For our e.g., suppose the base choices blocks are A, 1 and x. Then, the base choice test is (A,1,x) and the following additional tests are needed: (B,1,x), (A,2,x), (A,3,x), (A,1,y).
- A test suite that satisfies BCC will have one base test, plus one test for each remaining block for each partition. Totally, $1 + \sum_{i=1}^n (B_i - 1)$.



So, the first one is plainly called base choice coverage. What does it say? Say it is a base choice that is a partition of my choice is chosen for each characteristic, and a base test is formed by using the base choice for that characteristic.

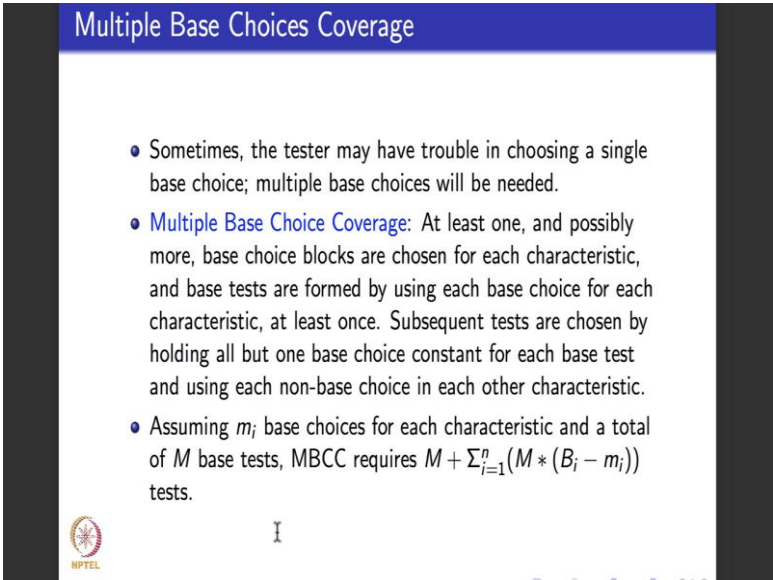
Once you form the base test you keep it aside. Subsequent test which means all the other remaining tests how are chosen? They chosen by holding all, but one base choice constant and using each non base choice in each other characteristic. So, just to explain it in a different term, what do we do in base choice? The focus is I say I want to focus on one partition one characteristic call it the base choice right. So, for example, the base choice could be A, 1, x. Take that as a TR keep it aside, then subsequent test how do I choose the test that I want to add on to it? I choose by holding all, but one base choice constant right and choose each non base choice in the other characteristics. For example, if you consider the same partitions that we had which was a, b, 1, 2, 3 and x, y. Let us assume that my base choice is 1 A, 1 x this 3 tuple A, ,1 x. So, the base choice test is this.

Now, what do I do? Other test that I choose to be like this. So, what do I do? I fix one and x instead of a, I replace it with b right. In the second case I fix A and x instead of one I replace it with 2 and 3; and in the third case I fix A and 1 instead of x I replace it with y, is that clear? So, what I do is I choose one class of partitions call it the base choice keep it aside. Then what I do is I fix, I do not vary the other base choices vary one base

choice. Like for example, once I have pick a one, x what are these remaining four tests? How do I obtain them after picking A, 1, x as my base choice. I park one and x, I vary A to B then in the second case I park A and x, I vary one to be 2 and 3 the other 2 blocks.

The third case I park A and 1 and I vary x to y. So, a test suite for base choice coverage how will I calculate how many tests are needed? If you see it will have one base test which like this which I have kept aside and then it will have one test for each remaining block of each partition that is what I have done here. So, totally the number of tests would be $1 + \sum B_i - 1$, where B_i is the number of blocks in each partition. Is that clear?

(Refer Slide Time: 21:36)



Multiple Base Choices Coverage

- Sometimes, the tester may have trouble in choosing a single base choice; multiple base choices will be needed.
- **Multiple Base Choice Coverage:** At least one, and possibly more, base choice blocks are chosen for each characteristic, and base tests are formed by using each base choice for each characteristic, at least once. Subsequent tests are chosen by holding all but one base choice constant for each base test and using each non-base choice in each other characteristic.
- Assuming m_i base choices for each characteristic and a total of M base tests, MBCC requires $M + \sum_{i=1}^p (M * (B_i - m_i))$ tests.

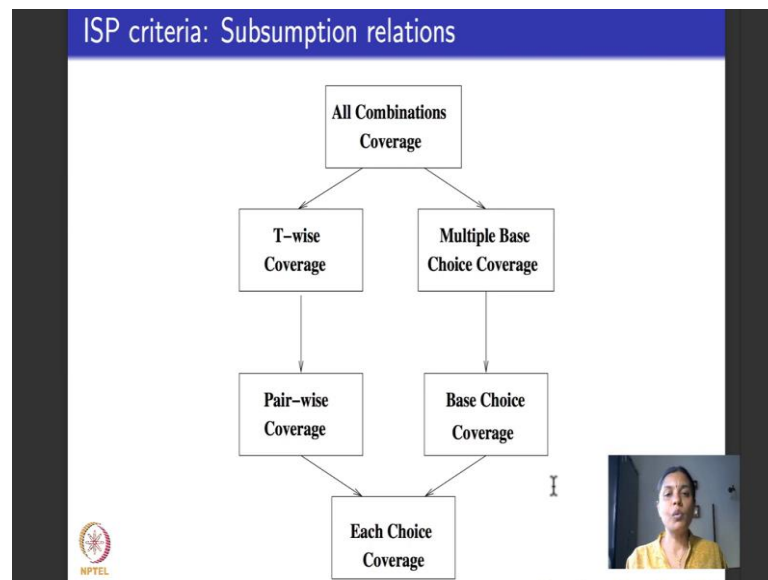
NPTEL

So, now moving on here we parked one base choice. There is nothing in that prevents from parking more than one partition as a base choice. If I park more than one partition of the base choice then I have what is called multiple base choices. So, what is multiple base choice do? Multiple base choices say that I have to pick up at least one, maybe more as base choice blocks for each characteristic. Then what do I do after that? Base tests are formed by using each base choice for each characteristic at least once.

So, that is like doing this A, 1, x where I had only one base choice. Here multiple base choices are there. So, you form you base test by using each of those multiple base choice in all the characteristics. After that what do I do? I do the same thing that I did for each base choice coverage. That is I hold all, but one base choice at any point in time constant and use each non base choice in each other characteristic.

For example, if you assume m_i base choices for each characteristic and the total of capital M base tests, then multiple base choice coverage MBCC abbreviated as MBCC requires how many tests? This m is kept aside and then after that one I fix my M I can do B_i minus M_i and sum it to overall. Is that clear?

(Refer Slide Time: 22:57)



So, this is how I do base choice and multiple base choice. So, what are the various coverage criteria that we have seen till now? We have seen all these six input space partitioning coverage criteria first one that we saw was all combinations coverage up here it was exhaustive testing with reference to partitions and all the characteristics. So, in terms of subsumption relation that subsumes all other coverage criteria. The next coverage criteria that we saw was each choice coverage criteria, which was the weakest of all the coverage criteria for input space partitioning because it just picks up one value from each of the partitions right and the choice of the values is completely random. So, it is quite a weak coverage criteria.

So, once I have pairwise coverage which means I park one value and let the others do vary and I consider them pairwise then I that extends each choice coverage criteria by definition and because T wise is any number assuming the T is any number greater than 2, T wise subsumes both pairwise and each choice coverage. In fact, as I told you T wise coverage will be equal to all combinations coverage if you consider T to be the cardinality of all combinations.

On the other side we do this base choice. It is like active clause coverage criteria on logic. I have fix one partition one characteristic as my base choice let the others vary. In plain base choice coverage, there is only one choice for the base in multiple base choice coverage there is more than one choice for the base. So, by definition multiple base choice subsumes single base choice I do not use the word single here, and there are no cross subsumptions here. Pairwise and base choice there is no relation, T wise and base choice there is no relation, similarly pairwise multiple base choice there is no relation and all combinations coverage definitely, because it is exhaustive testing subsumes multiple base choices and base choice coverage. Is that clear please?

So, what we will do in the next lecture is each of these coverage criteria I will take the triangle type example, and I will walk you through how the TR for each of these coverage criteria look like and how to write tests that satisfy test requirements for these coverage criteria.

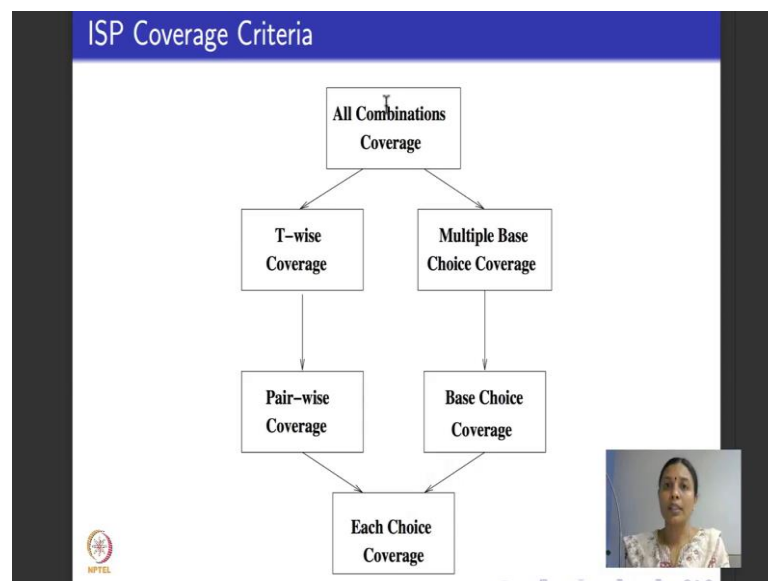
Thank you.

Software Testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 34
Input Space Partitioning Coverage Criteria: Example

Hello again, welcome to the last week lecture of 7. We will be done with input space partitioning this week. So, what did we do till now? I began the week with telling you about functional testing, we saw various popular categories of functional testing, then I moved on to looking at one particular core functional testing technique namely that of input of space partitioning. We saw various of partitioning inputs in the third lecture interface based partitioning in functionality based partitioning. In the last lecture I told you about coverage criteria that you could define based on input space partitioning.

(Refer Slide Time: 00:45)



So, we have 6 different coverage criteria. The exhaustive input space partitioning coverage criteria is what is called all combinations coverage or ACoC write up here, and the weakest is consider each choice of values from each partition of the blocks that is each choice criteria, it is arbitrary consider to be a weak coverage criteria. In between the popular one is pair wise coverage criteria where for every partition of the input space per pair of partitions, you park and then let the others vary- then you park another pair of partitions let the others vary.

This is supposed to be one of the most powerful input space partitioning coverage criteria that is used. An extension of pairwise would be T-wise coverage criteria where instead of working with a pair of partitions you work with T partitions at a time for a T that is greater than or equal to 2. The T happens to be all the partitions put together then we saw that it means nothing but all combinations criteria.

The ones on the left hand side where doing coverage arbitrarily without focusing on any particular characteristic or particular partition. We defined what is called base choice partition, which is the partition that would you like to focus on for some reason. And then we defined two coverage criteria based on the base choice partition: one is single base choice coverage also called base choice coverage. The next is multiple base choice coverage, where the number of base choices could be more than 1; you could choose more than 1 base choice coverage.



(Refer Slide Time: 02:36)

TriTyp example

Consider the interface-based ISP TR for the TriType example:

Partition	b_1	b_2	b_3	b_4
q_1 ="Relation of Side 1 to 0"	> 0	= 0	= 1	< 0
q_2 ="Relation of Side 2 to 0"	> 0	= 0	= 1	< 0
q_3 ="Relation of Side 3 to 0"	> 0	= 0	= 1	< 0

The above partitioning considers the relation of each side to 0 or 1, covers both valid and invalid inputs.

So, we saw all these 6 coverage criteria in the last lecture. We also took an abstract example and showed you how test cases or TR for each these of coverage criteria will look like. What I will do today is we will take one of the examples that we have done before in particular, we will take that triangle type example and then we will consider the partitions that we had defined in the third lecture for this week based on interface domain partitioning. And I will tell you how each of these coverage criteria can be defined on partitions for triangle type.

So, just to recap, triangle type was a program that took 3 sides of a triangle as input and it was defining the, classifying the triangle; whether it was not a valid triangle or it was a scaly in triangle or an isosceles triangle or an equilateral triangle based on the sides. So, one classic input space partitioning would be to consider what is the relationship of each side to, let us say, a 0 number or a 1 number.

Why is this kind of criteria important? It is important because if the side is less than 1 than we define invalid triangles and all other cases we defined various kinds of valid triangles. So, if you remember last lecture, I mean in the third lecture I had shown you this example for interface base inputs partitioning test requirement for that the TriTyp example. They were 3 criteria the first one said what is the relationship of side 1 to 0: is it greater than 0, is it equal to 0, is it equal to 1 or is it equal to 0. Similarly what is the relationship of side 2 to 0 and what is the relationship of side 3 to 0? Each these 3 cases we considered 4 partitions. So, we take side 1, side 2 and side 3 to be greater than 0, all 3 sides to be equal to 0, all 3 sides to be equal to 1 and all 3 sides to be less than 0.



So, these are 4 partitioning and it covers between them both valid and invalid triangles as we saw.

(Refer Slide Time: 04:12)

Possible test case values

One possible set of values that constitute test cases satisfying the partitions in the previous slide are:

Param	b_1	b_2	b_3	b_4
Side1	2	1	0	-1
Side2	2	1	0	-1
Side3	2	1	0	-1

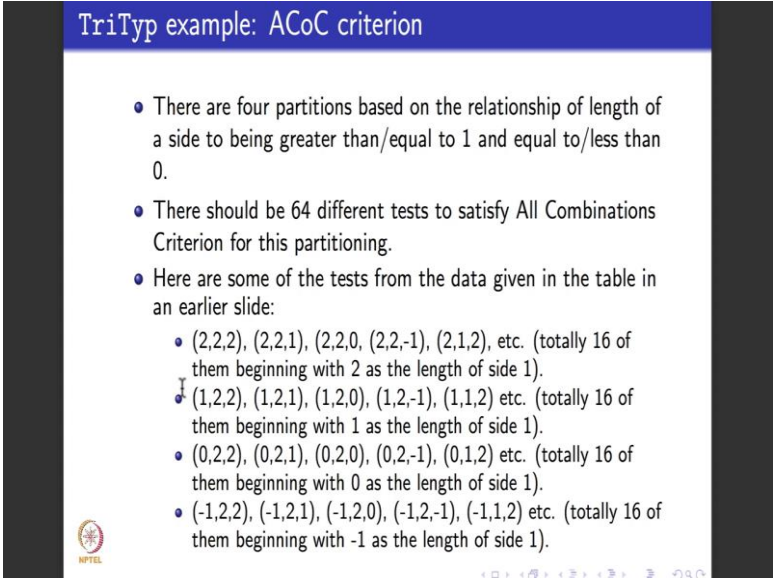
Now, what could be possible test cases for this test requirement? Here is one example value. So, let us say side 1 if you go back it is a choose values of side 1 greater than 1 0

equal to 0 equal to 1 less than 0. Similarly for side 2 and side 3. So, I have chosen value like that I have chosen side 1 to be 2, side 2 to be 1, side 3 to be 0, side 4 to be minus 1.

I could choose different values here, any different value, specially here for b 1 and b 4, but just for simplicity sake I have kept that also as 2 and minus 1. But any other number greater than 0 or any other number less than 0 will be good enough as values for the 3 sides it just as to meet criteria for this four partitions.

Now this is my set of test cases. On these set of test cases I am going to do go ahead and apply all these coverage criteria and see.

(Refer Slide Time: 05:10)



TriTyp example: ACoC criterion

- There are four partitions based on the relationship of length of a side to being greater than/equal to 1 and equal to/less than 0.
- There should be 64 different tests to satisfy All Combinations Criterion for this partitioning.
- Here are some of the tests from the data given in the table in an earlier slide:
 - (2,2,2), (2,2,1), (2,2,0), (2,2,-1), (2,1,2), etc. (totally 16 of them beginning with 2 as the length of side 1).
 - (1,2,2), (1,2,1), (1,2,0), (1,2,-1), (1,1,2) etc. (totally 16 of them beginning with 1 as the length of side 1).
 - (0,2,2), (0,2,1), (0,2,0), (0,2,-1), (0,1,2) etc. (totally 16 of them beginning with 0 as the length of side 1).
 - (-1,2,2), (-1,2,1), (-1,2,0), (-1,2,-1), (-1,1,2) etc. (totally 16 of them beginning with -1 as the length of side 1).

Suppose let me start with all combinations coverage. How many test case values are there, how many different partitions are there? 4 of them. How many inputs are there? 3 of them. So as per our formula how many different all combinations criteria test cases will be there?

There are 4 partitions that are based on relationship of length of a side being greater than or equal to 0 or one. So, what will be then total number of test cases the total number of test cases should be 64 right because for each of that I can choose any of 4 of them, any one of them, any of them keep going right. That is what the formula for all combinations coverage criteria it told us it is difficult for me to enumerate all the 64 test cases, but I have made an attempt to get you started if you want do that on your own. How do we go

about doing it? Keep an ordering, there is no need of ordering it is not enforced by the coverage criteria, but just to get your enumeration of this large number criteria 64 different cases right it helps to keep an ordering. So, what is what is the ordering that I have here if you see I have began with side 1 being as 2 and now I vary side 2 and side 3. So, the first one says all 3 sides 1, 2 and 3 are all value 2.

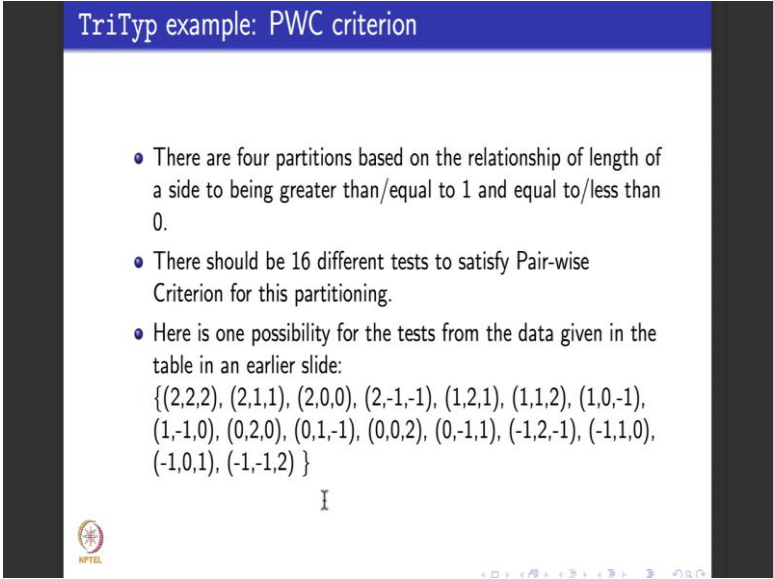
Remember in this listing, I have kept side 1 to be 2. Now, I vary side 2 to side 3 what are the values left out side 2 could be 2, side 3 could be one and side 2 could be 2, side 3 could be 0 and I move on, I say side 2 could be 2 side 3 could be minus 1. So, between these 4 cases what I have covered I have covered the option of side 1, side 2 being both 2 and side 3 ranging over the 4 different values 2, 1, 0 and minus 1. Now I change side 2, I say side 1 is still 2 side 2, let us say becomes one now you vary sides 0 for all these option. So, you start with 2 and now the next one in this list would which I have not written would be 2, 1, 1, the next one after that would be 2, 1, 0. The next one after that would be 2, 1 minus 1 I keep going like this. Four, four, four, each. So, totally I will get 16 of them beginning with side 2 as the length; length 2 as the length of side 1. Similarly let us a, consider length one as the length of side 1. I do the same thing I park side 1 as length one let us a 2 as length 2 and I vary side 3 first. So, here I make it side 3, 2, side 3, one side 3, 0 side 3, minus 1 and I move on.

I say side 1 is one side 2 is also one and I vary side 3. So, that will be 1, 1, 2 the next one in this list will be 1, 1, 1. The next one would be 1, 1, 0 and so on. So, here again I will get 16 of them beginning with one as a length of side 1. Now I say length of side 1 is 0 and again I do the same enumeration and I will get 16 more. Similarly length of side 1 is minus 1 I do the same enumeration I get 16 more. So, if I basically what does, it parks one number in this table varies across the other numbers parks.

The next number in the table there is at across the number and I will do this exhaustively that is what all combinations coverage criteria suppose to be. So, it result in 64 cases test cases. Please remember one more fact. Here when I talk about test cases I am just giving you the output a full test case you also have to give expected output, I have not given expected output I have just given the input. The complete test case you have to give the expected output also I prove I did not give the expected output I thought it is easier to explain just how the inputs vary based on the partitions.

So, moving on we realize that just for a small example like type of a triangle doing input space partitioning was effective enough this was good enough partitioning, but if you consider all combination criteria you end up with too many test cases. 64 of them is a large number of test cases we would really not want so, many of so let us move on and look at other coverage criteria.

(Refer Slide Time: 09:00)



TriTyp example: PWC criterion

- There are four partitions based on the relationship of length of a side to being greater than/equal to 1 and equal to/less than 0.
- There should be 16 different tests to satisfy Pair-wise Criterion for this partitioning.
- Here is one possibility for the tests from the data given in the table in an earlier slide:
 $\{(2,2,2), (2,1,1), (2,0,0), (2,-1,-1), (1,2,1), (1,1,2), (1,0,-1), (1,-1,0), (0,2,0), (0,1,-1), (0,0,2), (0,-1,1), (-1,2,-1), (-1,1,0), (-1,0,1), (-1,-1,2)\}$

So, now, I told you after that the most useful and important coverage criteria is pair wise coverage. So, for same triangle type example let us workout the test case inputs for pair wise coverage criteria. How will they look? As I told you as a recap there are 4 partitions based on the relationship of length of the side being greater than or equal to 0 and less than or equal to 0 or 1.

Now for pair wise coverage criteria if you apply the formula that we saw in the last lecture, how many different test cases should be there? There should be 16 different test cases for pair wise coverage criteria and as I showed you in the last example for that abstract partitioning there could be several different ways in which you could meet pair wise coverage. Here I have listed one possibility for tests for achieving pair wise coverage.

How do you read this? How do you understand it? The first 4 triples 3 triples in the set consider side 2 to be parked at number 2 and then it varies side length of sides one length of sides 2 and 3. So, side 1 is parked at the number 2, here one side 1 is to side 1 is to

here in the second one, side 1 is to here in the third one, side 1 is to here again in the fourth and then I vary sides 2 and 3 length.


So, here I made all 2 of them, side 2 and 3 both two, here I have made both one, here I have made both 0 and here I have made both minus 1 now what do I do at consider side 1 as 1. So, the next 4 in the list of that side 1 if you notice is one here in this one in this one in this one and in the eighth one and then I varies sides 2 and 3. I have made side 2 and 3, 2 and 1 here 1 and 2 here 0 and minus 1 here minus 1 and 0 here.

This is just my choice you could do anything it does not matter right. Several different test cases could be written to achieve pair wise coverage criteria. The next 4 that is from the ninth to the twelfth test case here park side 1 as 0 and then varies sides to 2 and 3. Here again the variations of sides 2 and 3 the numbers that you see a just my choice, feel free to substitute to the any other values for sides 2 and 3 that will equally well satisfied pair wise coverage criteria. And the last 13, 14, 15, and 16 side 1 is minus 1 and again sides 2 and 3 are varying as per one choice that I have given you. Here again you could feel free to substitute values for sides 2 and 3 in any way that is satisfy pair wise coverage criteria.

(Refer Slide Time: 11:44)

TriTyp example: MBCC criterion

- There are four partitions based on the relationship of length of a side to being greater than/equal to 1 and equal to/less than 0.
- We consider 2 and 1 to be base choices for side 1. This gives two base tests: (2,2,2) and (1,2,2).
- We get totally $2 \text{ (base)} + 6 + 6 + 6 = 20$ tests. Four of these are redundant. So, we get totally 16 tests.
- The tests are $\{(2,2,2), (1,2,2), (2,1,2), (2,0,2), (2,-1,2), (2,2,1), (2,2,0), (2,2,-1), (0,2,2), (-1,2,2), (1,2,1), (1,2,0), (1,2,-1), (1,1,2), (1,0,2), (1,-1,2)\}$.



So, for the triangle type program for input space partitioning given in this table with these as the values for pair wise coverage criteria you get 60, sorry, you get 16 different test cases which I given here. Now we will move on last time I introduced you multiple

base choice coverage criteria, but I did not really give you an example of what multiple base choice coverage criteria will be. So, I thought I will put that instead of doing those single base coverage criteria for this lecture. So, here is how the example for the MBCC criteria looks like. There are 4 partitions again the that reference table, what is side 3 sides related to 0 and 1. So, what I do I have many choices for my base choice? Let us consider for my multiple base choice coverage criteria the number of base choices to be 2 and I consider 2 and one to be base choices for side 1. This gives us 2 base tests right which will be 2 for side 1 which is 2 here, which is one here, and sides 2 and 3 would be 2 and 2 same.

Now, if you apply the formula for MBCC criteria at totally how many tests will I get? These 2 base tests that come from this above item here and then 6 more each combination. So, totally I will get twenty tests and if you try to enumerate all these twenty tests you will realize that 4 of them are redundant, they come as repetitions. I have removed the repetitions. I have directly given you the succinct reduced setup 16 different test inputs for this that satisfied multiple base choice coverage criteria. The 16 different test inputs are like this, these 2 are the 2 base choice test I have listed them here right up in the front and how are these obtained if you try to study them. What would you do here? Here again you will notice that side 2 is side 1 is 2 here and then I varied one and 2, 0, 2 minus 1, 2, 2, 1, 2, 0, 2, minus 1 now the next part I have put side 2 side 1 as 0 and then I have consider 2, 2 because these are my base choices right. So, I have park all and reduce the rest.

Similarly, here is minus 1, 2, 2. Similarly I do 1, 2, 1, 1, 2, 0, 1, 2, minus 1, 1, 2, 2, 1, 0, 2 and 1, minus 1, 2. So, for 2, 1, 1, I consider all options for 0 and minus 1, I consider only 2 options. So, totally there are 16 different test cases that will satisfy multiple base choice coverage criteria for the triangle type example. I hope this small exercise helps you to appreciate, how to do input space partitioning, and how to exhaustively list these test cases. Now these will be our actual test cases. To make them actual test cases what you have to do is to able up and expected output values. Like for example, if I take this first 3 case what is the triangle that I am looking at 2, 2, 2, all 3 sides are equal. So, I am looking at an equilateral triangle.

Now, what is the try type example suppose to give you as output for equilateral triangle? It was supposed to give a number put that as the expected output as the fourth tuple that

corresponds to the expected output that is how you arrive at a complete test case specification for each of these test cases.

(Refer Slide Time: 14:53)

Example: Infeasible partitions

Consider the following method that we had discussed earlier:

```
public boolean findElement(List list, Object element)
// Effects: If list or element is null throw NullPointerException
// else returns true if element is in the list, false otherwise
```

I

NPTEL

Now, moving on before I wind up I would like to make you understand one simple point. Like you know graph coverage criteria and logical coverage criteria for input space partitioning you can get infeasible test requirements. For example, if you remember let us revisit this find element method that we saw in the third lecture. If you remember I give you this specification of a find element method and because we are doing input space partitioning please remember we do not really need the code.

We just need to know what the method is what its inputs are and what its output are. So, it is enough to just give that much, This find element method that we discussed 2 lectures ago took a list of objects as input and then took an element as also an input and it basically checks if this element is present in the list. If it is present in list it is written true otherwise it is written false it is written exception if the list or the element is null.



(Refer Slide Time: 15:51)

Infeasible combinations of partitions

- Some combinations of partitions can be infeasible in the input domain model.
- For e.g., consider the boolean `findElement(list,element)` method with the following partitions:

	Partitions			
Characteristics	1	2	3	4
A: length and contents	one element	> than one, unsorted	> than one, sorted	> than one, all identical
B: match	element not found	element found once	element found more than once	–

- Invalid combinations are (A1,B3) and (A4, B2).



So, for this, for example, let us I have done input space partitioning. I have considered 2 different characteristics for partitioning my input. One characteristic that I have considered is the length and contents of the list. How long is this list comes as input? What are its contents what are the elements the constituents of the list? The second criteria that I have considered is there a match? It is a functionality based in input space criteria which means is the element found in the list it directly deals functionality of the concerned method.

So, with these 2 characteristics one is based on the length of the list and the kind of elements in the list. The second characteristic is based on whether the element is found in the list or not. Here are some example partitions. There are 4 partitions in the first case, 3 partition in the second case. So, the 4 partition in the first case are the length of the list is one, it has only one element, the length of the list is more than 1, the elements of the list are an unsorted order arbitrary order. The third partition says, the length of the list is more than 1, the elements of the list are in sorted order. The fourth partition says the length of the list is more than 1.

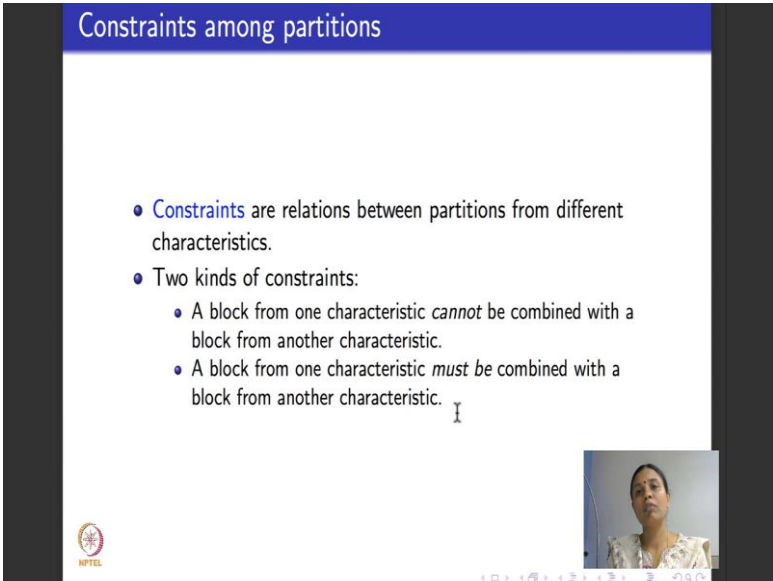
All the elements in the list are identical they are replicas of each other this is just one partition in based on this characteristic. For the second characteristic which I have called as match which basically test whether the element is present in the list or not, there are 3

partitions. The first partition says that element is not present in the list second partition it says element is present in the list, but only once.

Third partition says the element is present in the list more than once. This is some partitions that I have come up with. My goal is to be able to test this method based on these partition test cases obtained from these partitions. Now before moving on you will realize that some combinations of partitions are not valid, they are infeasible. For example, if you see the combination a, 1 and b, 3 what is a, 1 a one is this a is here, 1 is here, a, 1 is the list has exactly 1 element. What is b, 3 ? b is here which is about the match 3 says element found more than once. Clearly, a, 1 says- the list has exactly one element it is a length one b, 3 says the element is found more than once. These 2 cannot be coexist because of the list is of length one how can the element be found more than once. So, it is impossible to write a test case for the combination a, 1, b, 3. So, it is an infeasible combination.

Similarly, a, 4 and b, 2 is also an infeasible combination. What does a, 4 say? a and 4 it is says the for length of list is greater than 1 all the elements in the list are identical this list consisting of same elements copied several times. What does b 2 say? It says that the element is found in the list and it is found exactly one. Clearly, if the length of the list is more than 1 and the element copies itself again and again in the list the element cannot be found once; so this is another infeasible coverage criteria.

(Refer Slide Time: 19:13)



Constraints among partitions

- **Constraints** are relations between partitions from different characteristics.
- Two kinds of constraints:
 - A block from one characteristic *cannot* be combined with a block from another characteristic.
 - A block from one characteristic *must be* combined with a block from another characteristic.

NPTEL

Video inset showing a presenter.

So, when we design partitions we have to be careful if it is quite natural that we will get infeasible coverage criteria we have to be able to rule them out. So, constraints capture these infeasibility amongst partitions. What are constraints? Constraints are relations between partitions from different characteristics like we saw here right they cannot be a relation when the element list has more than 1 element all identical and you say the element is found only once it cannot be a relation at all. So, there are 2 kinds of constraints one which says that a block from one characteristic cannot be combined with a block from another characteristic. The example that we saw in these slides belongs to the first category. The second category which I have not shown in that example says that a block from characteristic must be combined with the block one characteristic.

If you go back here for the same example for example, I could say when the list has exactly one element then it must be combined with the element not found or with the element found more than, found exactly once. It cannot be combined with an element found more than once.

(Refer Slide Time: 20:29)

ISP coverage criteria and constraints among partitions

- For ACoC, PWC and TWC, the only option is to drop the infeasible combinations from consideration.
- Constraints can be handled better with BCC and MBCC criteria. The base case(s) can be altered to handle infeasible constraints.

So, cannot and must, these 2 are what the constraint say. You must combine some partitions with some partitions to make sense for test cases you should not combine some partitions with certain other partitions to make sensible test cases that are actually feasible. So, what do we do for all combinations coverage criteria, pair wise and T wise coverage criteria, you really cannot do much, you just have to drop the infeasible test

requirements. Why that is so, because, if you remember these coverage criteria do not have any intelligence they just blindly combine the partitions. And if partitions are infeasible there is nothing you can do about it, just drop them out. But if you have base choice coverage criteria or multiple base choice coverage criteria then may be you need not to choose base choices as those partitions that result in too many infeasible test requirement or test cases. So, you have better hold of infeasible test requirements by choosing the choice of your base choice to handle or minimize the infeasible test requirements appropriately. So, while partitioning the input and while writing test cases based on input base partitioning, please remember that sometimes you will get infeasible test requirements and you might have to omit that.

So, next week we will move on to a completely different module and testing called mutation testing and this will be the end of input space partitioning for you.

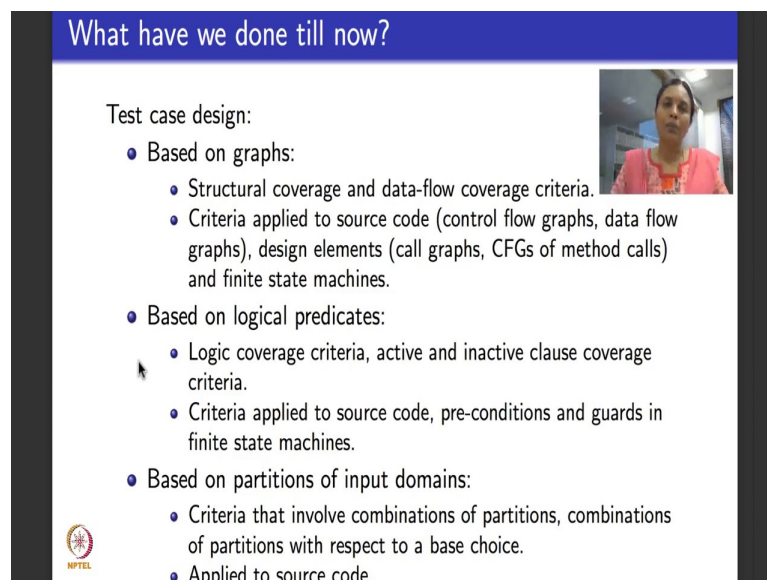
Thank you.

Software testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 35
Syntax-Based Testing

Hello again, welcome to week 8 we begin the first lecture of week 8 in this series. Before we move on and look at syntax based testing which is going to be the topic for some time in this week and next week, I would like to spend some time because we are more than halfway through the course it helps to recap and see what we have done till now. What have we done, what have we achieved, what have we understood?


(Refer Slide Time: 00:22)




What have we done till now?

Test case design:

- Based on graphs:
 - Structural coverage and data-flow coverage criteria.
 - Criteria applied to source code (control flow graphs, data flow graphs), design elements (call graphs, CFGs of method calls) and finite state machines.
- Based on logical predicates:
 - Logic coverage criteria, active and inactive clause coverage criteria.
 - Criteria applied to source code, pre-conditions and guards in finite state machines.
- Based on partitions of input domains:
 - Criteria that involve combinations of partitions, combinations of partitions with respect to a base choice.
 - Applied to source code.



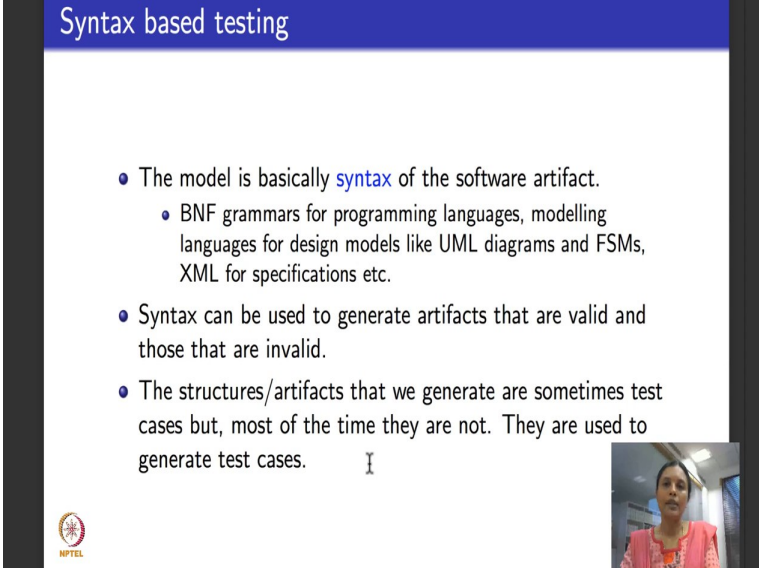


So, after the first week where I introduced you to motivation and initial terminologies of testing we have mainly focused on algorithms and methodologies for test case design. What did we do? We first draw graph based test case design, I introduced you to structural coverage criteria, data flow coverage criteria. Then we took various software artifacts source code, do the control flow graph, data flow graph with the source code, so how to apply the coverage criteria on them. Then we looked at call graphs, so how to apply the coverage criteria on them. Then we looked at CFG's and finite state machine models for requirements and so how to apply coverage criteria on them. There is a nice tool graph coverage web app that is available as a part of the course page.

I have also spent one lecture introducing you to the traditional classical terminologies that people use for source code testing and also for design element testing as a part of these weeks. We also recap-ed some algorithms on graphs depth first search, breadth first search, algorithms to enumerate prime paths and so on. After that exhaustive module on graph base coverage, we went on to coverage based on logical predicates. There again we saw one exclusive module recapping logic predicates clauses what they are all, so coverage criteria based on predicates and clauses these assumption active inactive clause coverage criteria and so on. Then we saw how to apply to source code, preconditions, guards that common finite state machines and so on.

Finally, last week, we took black box testing we took input domain, input space partitioning, saw how to design test cases by partitioning the inputs in various ways and how to apply to testing a typical software artifacts like code. What are we going to do on from now on in the course? We are going to do what is called syntax based testing in the course.

(Refer Slide Time: 02:25)



The slide is titled "Syntax based testing" in a blue header. It contains three bullet points:

- The model is basically **syntax** of the software artifact.
 - BNF grammars for programming languages, modelling languages for design models like UML diagrams and FSMs, XML for specifications etc.
- Syntax can be used to generate artifacts that are valid and those that are invalid.
- The structures/artifacts that we generate are sometimes test cases but, most of the time they are not. They are used to generate test cases.

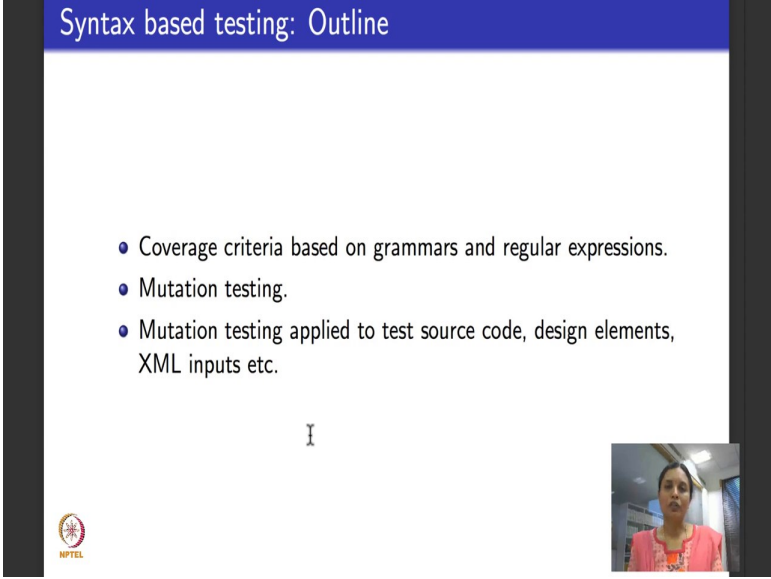
In the bottom right corner, there is a small video inset showing a woman speaking. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.

Typically almost every software artifact the deal we deal with be source code design or specification has an underlying syntax which tells you what are the building blocks of that software. Like for example, if I take C program I know that you to be able to write an if statement I have to do such a such thing, to be able to write a while statement I have to do such a such thing, I always have to call the main function and so on right.

So, it tells me what we can do or how to write programs, which are valid programs syntactically, which are invalid program syntactically. Typically syntax for programming languages are given as some form of grammars. We will look at them in detail. You also have syntax for modelling languages like UML diagrams, finite state machines that come as a part of UML diagrams and then, many software these days take inputs as XML right, XML is eXtensible Markup Language which defines an input format for several different kinds of entities. That also has a very well defined syntax.

Every software in artifact some kind of syntax and what we are going to see is, can I exploit a work with the syntax of the software to be able to test the software. You can use the syntax to generate artifacts that are valid, that are invalid and we will see how to work with valid and invalid syntactical software entities to be able to test them or write test cases for them.

(Refer Slide Time: 03:53)



The slide is titled "Syntax based testing: Outline" in a blue header. It contains a bulleted list of three items: "Coverage criteria based on grammars and regular expressions.", "Mutation testing.", and "Mutation testing applied to test source code, design elements, XML inputs etc.". In the bottom right corner, there is a small video inset showing a woman in a pink top. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.

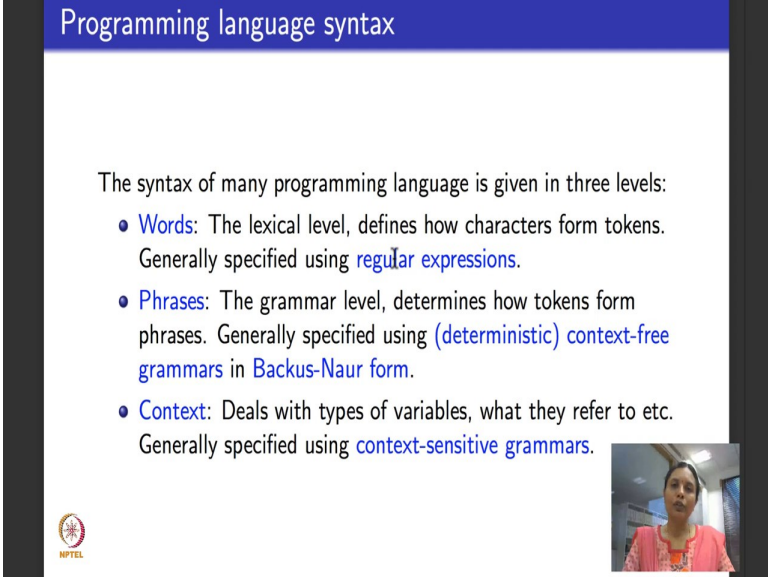
- Coverage criteria based on grammars and regular expressions.
- Mutation testing.
- Mutation testing applied to test source code, design elements, XML inputs etc.

So, what will be the outline of this? So, we will, this whole of this week and well into next week which is week nine, we will only be dealing with syntax testing. So, we will as you always look at coverage criteria based on two different entities: context free grammars and regular expressions I will introduce both of them to you briefly in this lecture and also discuss coverage criteria. Then in the next lecture I will teach you a term called mutation testing which basically mutates or changes the syntax of a program. Then we will look at test cases coverage for mutation testing, how to do mutation testing for

source code and also for design elements. There are specific mutation operators available when you integrate methods for object oriented call coverage, we look at all of them. And finally, we will see mutation operators for a markup language like XML which is very popular which basically tells you how to manipulate the inputs to a program.

So, the focus of this lecture will be the first one here. I will tell you what grammars are, what regular expressions are, how they come as syntax of programming languages, how to define coverage criteria on them and test them.

(Refer Slide Time: 05:04)



The slide is titled "Programming language syntax" in a blue header. The main content area is white and contains the text: "The syntax of many programming language is given in three levels:". Below this, there is a bulleted list with three items: "Words", "Phrases", and "Context". Each item is preceded by a blue dot and followed by a description of its level and how it is specified. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it. In the bottom right corner, there is a small video inset showing a woman with dark hair wearing a pink top, speaking.

Programming language syntax

The syntax of many programming language is given in three levels:

- **Words:** The lexical level, defines how characters form tokens. Generally specified using **regular expressions**.
- **Phrases:** The grammar level, determines how tokens form phrases. Generally specified using **(deterministic) context-free grammars** in **Backus-Naur form**.
- **Context:** Deals with types of variables, what they refer to etc. Generally specified using **context-sensitive grammars**.

NPTEL

So, typically if you understand a bit of how compilers work, compilers check if a program is syntactically fine. They compile the program and generate object code. So, every programming language that is meant to be compiled or interpreted comes with an underlying syntax. How is the syntax of a typical programming language defined? Here is how it is defined, it is typically defined in three different levels. To start with there are what a count words which constitute the lexical level and they tell you how to define words from various characters, how the various characters form tokens.

Generally words are specified using what are called regular expressions which are expressions that constitute regular languages and can be accepted by finite state automata. You should, if you have done a course on automata theory, you would know what regular expressions are. At a next level after words in programming languages come what are called phrases. Phrases are usually defined using a context free grammar

that is given in the particular normal form called Backus-Naur form. In fact, it is not an arbitrary context free grammar, it is what is called a deterministic context free grammar. We will tell you what a CFG is in this lecture. And then, at this level, the grammar level phrases determine how tokens form phrases. After these we have what is called context information or context sensitive information which cannot be explained, expressed when we do phrases. That deals with variables, objects, their type what do they refer to and so on and context is generally specified using context sensitive grammars.

So, three levels generic levels and which syntax many programming languages are defined, words given as regular expressions, phrases given as a context free grammar in a particular normal form called Backus-Naur form followed by context which is specified using context sensitive grammars. When we deal with mutation testing we are going to mainly deal with words and phrases because that is what you can manipulate syntactically really well and get variants of a software program or variants of an input. So, when we deal with words and phrases because their expressed using regular expressions in context free languages, it helps to recap what regular expressions and context free languages are. That is what I am going to do for the rest of this lecture for you.

(Refer Slide Time: 07:21)



Regular expressions

- Syntax of regular expressions over an alphabet A

$$r ::= \emptyset \mid a \mid r + r \mid r \cdot r \mid r^*$$

where $a \in A$.
- Semantics: associate a language $L(r) \subseteq A^*$ with regexp r .

$L(\emptyset)$	$=$	$\{\}$
$L(a)$	$=$	$\{a\}$
$L(r + r')$	$=$	$L(r) \cup L(r')$
$L(r \cdot r')$	$=$	$L(r) \cdot L(r')$
$L(r^*)$	$=$	$L(r)^*$

So, we will begin with regular expressions I will also tell you what context free languages are. If you want to know more details feel free to pick up any good book on

automata theory or theory of computation you will get to know more about regular expressions and context free grammars. Feel free also to ping me. In fact, there are NPTEL courses on automata theory that are available you can also write to me for any doubts about these.

This is a one off module like we did graph algorithms that you would need to understand syntax based testing a little more thoroughly. So, syntax of a regular expression over an alphabet A . Alphabet A is an alphabet that defines the building block of the programming language or very much like the alphabet of a natural language that we look at. Once you have an alphabet A the regular expressions are defined using the syntax. You say the empty expression given by ϕ like this is a regular expression. Every letter there small a belongs to the alphabets at capital A , single letter is a regular expression on its own. Inductively, if I have regular expressions r and another regular expression also denoted by r , please remember this r and this r need not be the same regular expression, then their plus r plus r is another regular expression. Similarly, given two different regular expressions r into r they could be same they could be different r dot r or r concatenated with r is another regular expression.

Given a regular expression r , r^* is another regular expression. So, what is syntax? It says the empty set is a regular expression. Every single letter from the alphabet is a regular expression. Addition or plus or union of two regular expressions is another regular expression. Concatenation of two regular expressions is another regular expression, star of a regular expression is another regular expression this defines a syntax. What do these operators mean? Each of these regular expressions defines what is called a language over the alphabet A . The set of all words over the alphabet A is denoted by A^* and each regular expression r defines a language L of r which is a subset of A^* . The language associated with empty regular expression ϕ as you would expect is the empty set. There is nothing, it is just an empty set. Language associated with the regular expression which is just a letter from the alphabet is this set containing the single letter word which is just that letter from the alphabet, the same letter from the alphabet.

If I have languages associated with regular expressions r and r' inductively called L of r and L of r' then the language associated with the expression r plus r' is given by the union of the two languages corresponding to r and r' . The language associated with the regular expression r into r' is given by the concatenation of the

language associated with r , which is L of r and the language associated with r prime which is L of r prime. The language associated with the regular expression r star denoted by L of r star is, you take the language associated with r and do the star operation. I hope you are all familiar with how to take union of two languages, it is a normal set theoretic union. What is the dot of two languages? Dot is a juxtaposing or concatenation. Suppose I have a word w from the first language, another word w prime from the second language L of r prime, then w dot w prime is you take w and append that w prime and the end of w you can concatenate or juxtapose w and w . L of r into L of r prime is take every word from L of r and concatenate with every other word from L of r prime and this whole language is the concatenation of the two languages.



For star, star is 1, 0 or more concatenations of the same language, right.

(Refer Slide Time: 11:06)

Examples of Regular Expressions

Expressions built from a, b, ϵ , using operators $+$, \cdot , and $*$.

- $(a^* + b^*) \cdot c$
"Strings of only a 's or only b 's, followed by a c ."
- $(a + b)^* abb(a + b)^*$
"contains abb as a subword."
- $(a + b)^* b(a + b)(a + b)$
"3rd last letter is a b ."
- $(b^* ab^* a)^* b^* \text{ } \mathbb{I}$

So, now let us look at some examples to understand how regular expression mean? The first one I have answered it for you. What is it say? Here is a regular expression there are three letters in the alphabet a, b and c and what I have done here - I have used a star plus b star concatenated with c . If you go back and look at the syntax, I have used all the three operations I have used plus I have used dot and I have used star. Plus means take union, dot means concatenate or juxtapose star is the Kleene star. So, what is this regular expression mean? It means take any number of occurrences of a because a star is 0 or more occurrences of a and take any number of occurrences of b . It could be single a , two

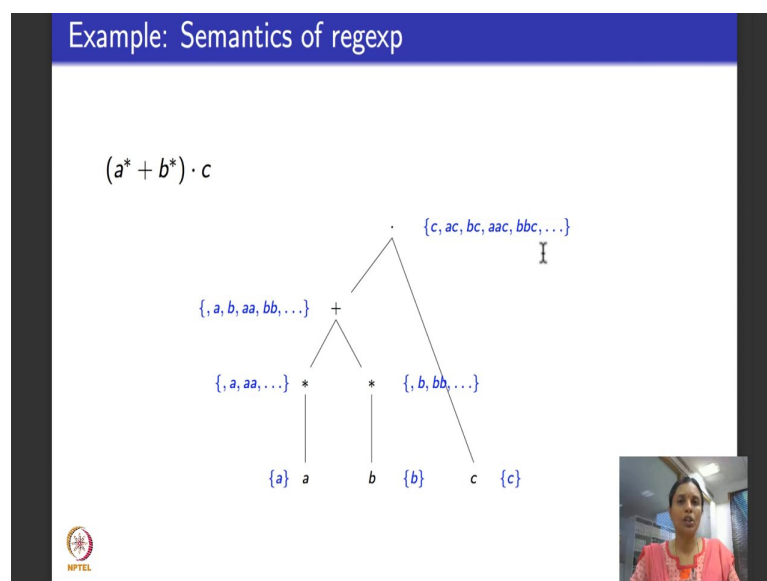
a's, three a, say it could be single b, two b, three b's and so on. Take the union of these two sets which is the plus and concatenate it with the single letter c. So, it means it is strings of only a's or strings of only b's that end with a c that are followed by c. So, the language defined by this regular expression is this language is this string of only a's or only b's that are followed by a c.

Let us look at another example here is another regular expression is says a plus b the whole star, in the middle you have abb strictly speaking you should write it as a dot b dot b I have simplified it because I do not have to all the time write the dot notation, it is a standard practice, followed by another a plus b. So, what it says this that a plus b means it is can be either a or b star means it can be any word that involves a or b any combination of a or b here again at the end its any combination of a or b, but in between they must be a pattern for sure what is that pattern that battle is abb. So, this is the set of all words that contain abb as a sub word in that this is the regular expression.

Now can you tell me what this regular expression we will stand for? I will read it out for you. It says a any number of any word over a and b, in between there is ab and then this is one occurrence of a or b followed by one more occurrence of a or b. Which means the last letter could be an a or a b, the second last letter could be an a or a b, the third last letter from the right is only a b and then they could be any word over a and b in the front. So, the language for this is, is the set of all words where the third last letter in the word is a b.

So, now, again here is another expression. This does not involve plus for a change it only involves dot and star, I will read it out for you. It is b star a b star a the whole star followed by b star. Spend of few seconds thinking about what the language that it could define. This b star means that it could end with a b and this, what is this mean? This means that if there is a word that is not epsilon then there is one a in it for sure and one more a in it and the rest could be or b a need not be there. So, the language that is defined by that what would it be, think about it and if you do not know ask me in the forum and I will tell you.

(Refer Slide Time: 14:24)



Now, how these are how regular expressions look like. Remember why are we looking at regular expressions because we want to use them in the context of a programming language syntax, we are in level one here. So, programming language syntax there is something called a lexical analyzer which parses a regular expression and generates some entities out of them. So, how does that parsing happen? I will illustrate it using an example.

So, suppose I had this regular expression which was the first in the list that we saw here. $a^* + b^* \cdot c$, all words ending with the c . If I parse this this is how I get; how do I read this. Read this is a binary tree the leaves of the tree or all letters from the alphabet, the internal nodes could have one child or two children. If it has one child and it is a unary operator like star. Like for example, read this part as b^* , the star operation applied to b , this is a star, the star operation applied to a and then I take a^* and b^* and add it that is this part in the regular expression and then what do I do, I take separately I concatenate it with a^* and b^* . This is how the tree corresponding to this regular expression looks like and almost all regular expressions you can write such trees.

How do I define what is the language? With every node in this tree, I can associate a language and I inductively work my way build my way up in the tree to get the language corresponding to the entire regular expression. Like for example, for these leaf nodes by

our semantics, the language singleton a singleton b singleton c for this is star. So, it is 0 or more occurrences of a, this is b star 0 or more occurrences of b. I am sorry there seems to be a small typographical error, they should be epsilon here before the comma and another epsilon here before this comma and after that this is plus. So, it is all words that have only occurrences of a, only occurrences of b and after this is dot which is all words that end with the c. So, this is how you get the semantics of a regular expression.

So, this was just brief quick introduction to regular expressions. In fact, language accepted by regular expressions are called regular languages. We also have automata models for them, finite state automata which we saw earlier in the course exactly correspond to languages that are defined by regular expressions called regular languages. There is a theorem called Kleene's theorem, a theorem which let us you convert a given regular expression into a finite state automata and vice versa, any book on automata theory we will help you to understand that better. So, that was a brief introduction to the first part of the grammar which is this words. Now I will move on and tell you what context free grammars are and how are they defined.

(Refer Slide Time: 17:12)

Context-Free Grammars: Example 1

CFG G_1

$$\begin{aligned} S &\rightarrow aX \\ X &\rightarrow aX \\ X &\rightarrow bX \\ X &\rightarrow b \end{aligned}$$

Derivation of a string: Begin with S and keep rewriting the current string by replacing a non-terminal by its RHS in a production of the grammar.

Example derivation:

$$S \Rightarrow aX \Rightarrow abX \Rightarrow abb.$$

Language defined by G , written $L(G)$, is the set of all terminal strings that can be generated by G .

What is the language defined by G_1 above? $a(a+b)^*b$

So, here is an example of a context free grammar. So, what does this say, how do I read this? There is a special symbol called S , S stands for start symbol and then there are these production rules. Each production rule has a left hand side and a right hand side. The idea is, you begin with the start symbol keep applying the production rules till you

cannot apply them anymore. What do we mean by applying a production rule? Start with S, replace the symbol S by using one of the rules that are available with the right hand side. So, for S this is the only production rule available in this grammar which is from S you can get aX. So, I have replaced S with aX using this rule. Now I have X.

There are three other production rules that are available with X. So, I could replace X with an aX, replace this X with the bX or replace X with a b. So, let us say I chose to replace X with the bX that is I have used the third rule in this list. I choose to now replace this X with just this b. So, now, you see there are no more of these capital letter symbols available and I, no more production rules to replace anything with this, so I stop. When I stop I say this is the word that is generated by the grammar. So, grammar, this context free grammar generates this word abb.

So, in general how do you write the language defined by the context free grammar? It is the set of all words or terminal strings. why are they called terminal strings? Because there is nothing, they cannot be replaced further. All the terminal strings that are generated by the grammar constitute the language given by the grammar.

So, these letters in capital alphabet, the symbols in capital letters they are what are called non terminals the symbols in small letters they have what are called terminals. So, now, this grammar I just gave you one word. What would be the language defined by this grammar, can you think of it? In fact, if you realize that the I can once I start do X as aX right. So, all letters begin with a, all the words begin with a because this is only rule that I can apply to start with and then I can replace the X with aX, bX. Here I can go on replacing I generate one more copy of X to replace it. At some point I say I am going to stop like what I did here I choose this last rule where I replace the X with a b. So, this grammar defines all words that begin with an a and end with a b and this is the context free language. So, I have written it like a regular expression, all words that begin with an a and end with a b.

(Refer Slide Time: 19:54)

Context-Free Grammar

A Context-Free Grammar (CFG) is of the form

$$G = (N, A, S, P)$$

where

- N is a finite set of non-terminal symbols
- A is a finite set of terminal symbols.
- $S \in N$ is the start non-terminal symbol.
- P is a finite subset of $N \times (N \cup A)^*$, called the set of productions or rules. Productions are written as

$$X \rightarrow \alpha.$$

So, in general a context free grammar looks like this. It has four entities, there is a set called capital N which is a finite set of non terminal symbols. If you go back here then capital N, the set capital N consists of the two non terminals S and X. There is a set a of terminal symbols we call it alphabet or sigma, then there is one designated non terminal in the set N called starts non terminal symbol denoted always by S and then P is a finite set of production rules. What is the format of the production rules? Production rules always look like this. On the left hand side there is exactly one non terminal which comes from this set and on the right hand side, it could be any string that comes from N union a star. That is it could be any string that is a combination of non terminal symbols from N and terminal symbols from a.


So, it is a member or an element of N cross producted with N union a star which I for convenience and readability stake write like this. So, this corresponds to this capital N, it is a member of that set, this alpha is a string from this set. Is that clear please?


(Refer Slide Time: 21:08)

Derivations, language etc.

- “ α derives β in 0 or more steps”: $\alpha \Rightarrow_G^* \beta$.
- First define $\alpha \xrightarrow{n} \beta$ inductively:
 - $\alpha \xrightarrow{1} \beta$ iff α is of the form $\alpha_1 X \alpha_2$ and $X \rightarrow \gamma$ is a production in P , and $\beta = \alpha_1 \gamma \alpha_2$.
 - $\alpha \xrightarrow{n+1} \beta$ iff there exists γ such that $\alpha \xrightarrow{n} \gamma$ and $\gamma \xrightarrow{1} \beta$.
- **Sentential form** of G : any $\alpha \in (N \cup A)^*$ such that $S \Rightarrow_G^* \alpha$.
- Language defined by G :

$$L(G) = \{w \in A^* \mid S \Rightarrow_G^* w\}.$$
- $L \subseteq A^*$ is called a **Context-Free Language** (CFL) if there is a CFG G such that $L = L(G)$.





So, what is the language corresponding to a context free grammar, language corresponding to a context free grammars defined like this. You start from the start symbol, keep applying production rules one or more times as long as you have a non terminal symbol to replace and after some N applications of production rule when you cannot apply them any more you would get a string full of terminals and that is where you stop.

So, how do I define that? I say alpha derives a string beta in zero or more steps, zero or more we always denote it by using the star in the grammar G. How do I define it? I define it by induction on the number of steps let us say N is the number of steps. You say alpha derives beta and N steps if either alpha derives beta and one step over alpha derives beta and N step and then I say how alpha derives beta in N plus 1 steps. How do I say alpha derives beta in one step? If alpha is of this form alpha one X alpha two and there is already a production rule in the grammar available which let us you take this non terminal symbol X and replace it with a string gamma, which means what in the string alpha 1 X alpha 2, I take X out and using this production rule replace the place where X was there with gamma.

In other words I derive alpha gamma, alpha 1 gamma alpha 2 from alpha 1 X alpha 2 by using the production rule X goes to gamma and I keep doing this one step, one step. Inductively, how do I say alpha derives beta and N plus 1 steps? If there exist some


intermediate string gamma such that alpha derives gamma and N steps and from gamma in one step I can obtain beta, is that clear please.

Now, these intermediate entities that you see alpha, gamma and all this is a term in grammar for them, they have called sentential forms. Ultimately, so if I go back to this example, this is sentential form, this is a sentential form. Every step in the derivation leads to an entity called sentential form and finally, sentential form end in a string of terminals that belongs to the language generated by the grammar. So, the language generated by the grammar is a set of all words over the alphabet of the grammar such that from the start symbol S, I can derive using the rules of the grammar as explained here, the word w. Language generated by such grammars context free grammars are called context free languages. You might wonder why the term context free? For that if you go back here you will understand. Production rules in context free grammars are always up this form, I take a non terminal X, I replace it with symbol alpha.


This is irrespective of the context in which X occurs. Like for example, when I use it here I say X occurs here somewhere in between my sentential form. Without knowing what alpha 1 and alpha 2 is I can directly plug in a rule and replace X with gamma. So, when I do that I am free of the context in which X occurs and that is why it is called context free grammars.

(Refer Slide Time: 24:12)

Regular expressions and grammars



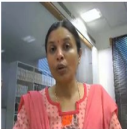
- Although regular expressions are sometimes sufficient, a more expressive grammar is often used.
- Also, regular expressions can themselves be written as grammars.
- We will deal with coverage criteria for grammars.



So, typically regular expressions and grammars are described by using grammar notation when syntax of programming languages are written. So, we really do not write them as two steps, we combined the first two steps and write it as a grammar that defines the combination of both the steps put to together.

(Refer Slide Time: 24:31)

Another example




```

stream := action*
action := actG | actB
actG   := "G" s n
actB   := "B" t n
s      := digit1-3
t      := digit1-3
n      := digit2 · digit2 · digit2
digit  := 0|1|2|3|4|5|6|7|8|9

```

Some of the strings generated by the grammar are G 17 08.01.90,
B 13 06.27.94 etc.



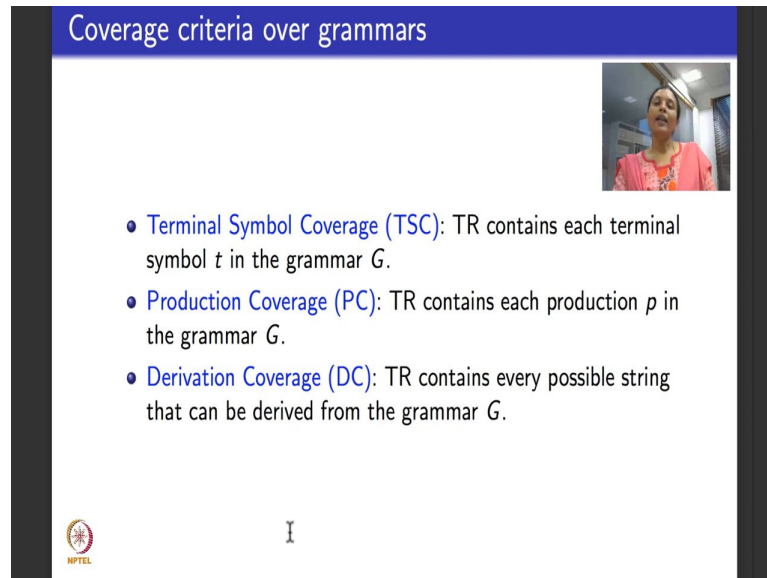
So, here is an example suppose this is how a grammar will look like here it if is notice it will have notations from regular expressions and notations from grammars.

So, I start with what is called a stream. A stream could involve one or more actions denoted by action star and action could be a non terminal of the form act G or a non terminal of the form act B. Read this is having two different production rules. One which says action is act G and action is act B, act G has this format it could be G s n, G is like a terminal string parked, that is why I have put it within double codes. Act B is B t n, B is another terminal string, I have put it within double codes. s and t could be a digit of a single unit or three unit it is a digit of length 1 to 3 denoted like this digit to the power of 1 to 3; t is another digit of length 1 to 3, n is has this pattern - how do I read this it is a two digit number denoted by digit to the power of two concatenated with another two digit number concatenated with another two digit number. And what could be a digit? Digit could be anything from 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, right.

So, if I take this grammar how we will I generate strings? I start with stream and then I do one of these act of G or act of B based on what I do the string begins with a G or with

B as indicated here it begins with a G or with B. Once I have a G or B, I could have two or three digit number like I have 13 here and 17 here and then I have this, two digits followed by a dot followed by two digits followed by a dot followed by two digits which come from this rule for n. So, string generated by this grammar out of this form, is that clear?

(Refer Slide Time: 26:25)



Coverage criteria over grammars

- **Terminal Symbol Coverage (TSC):** TR contains each terminal symbol t in the grammar G .
- **Production Coverage (PC):** TR contains each production p in the grammar G .
- **Derivation Coverage (DC):** TR contains every possible string that can be derived from the grammar G .

HPTEL

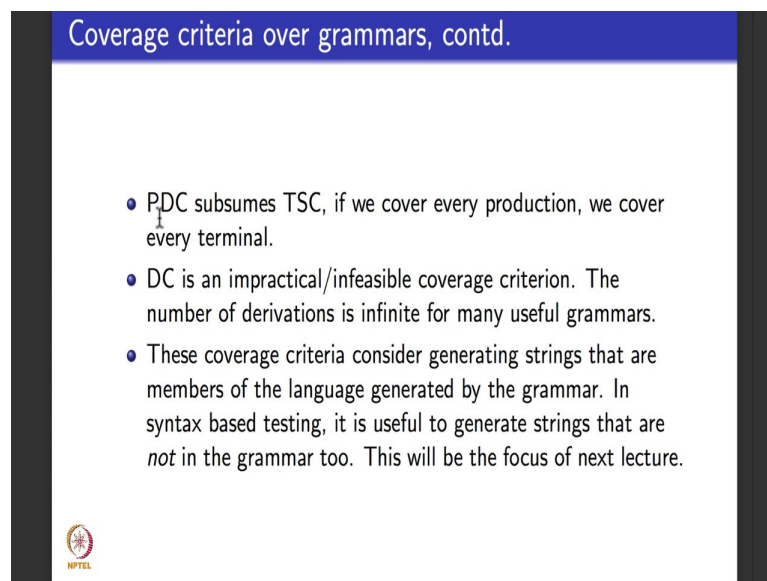
So, now I again define coverage criteria over grammars. So, what do grammars contain? Grammars contains terminal symbols right, non terminal symbols, start symbol and productions. So, basically the rules say cover each of them, cover every terminal symbol called terminal symbol coverage TSC. Test requirement contains each terminal symbol t in the grammar G . Cover every production rule which is production coverage. TR contains each production p in the grammar G and cover every derivation, DC derivation coverage. TR contains every possible string that can be derived from the grammar G .

We have not included a non terminal coverage because if you think about it you will realize it does not make much sense. So, we define three kinds of coverage criteria directly over plain grammars. We will see how to apply to grammars of various things like XML, programming languages and all. These three coverage criteria like we did for other things just say cover every terminal, cover every production, cover every derivation.

If you think about it terminal symbol coverage and production coverage make sense because terminal symbol says that every symbol must be generated by the grammar in the alphabet otherwise there is no point in the symbol being there in the alphabet. Production rule coverage also makes sense because if you do not use a production rule then it might as well not be there, but derivation coverage is an infeasible test requirement. Why is that so, because there could be infinitely many derivations right? As long as there is a rule like this in the example that we solve which let us you take X and give back X, you could replace this, use this rules several times again and again and again each one is a derivation and result in infinite number of strings.


So, derivation coverage can be sometimes difficult because it says test requirement is derive every possible string. Typically grammars generate infinite languages and it is not possible to derive every possible string, you will not terminate.

(Refer Slide Time: 28:23)



Coverage criteria over grammars, contd.

- PDC subsumes TSC, if we cover every production, we cover every terminal.
- DC is an impractical/infeasible coverage criterion. The number of derivations is infinite for many useful grammars.
- These coverage criteria consider generating strings that are members of the language generated by the grammar. In syntax based testing, it is useful to generate strings that are *not* in the grammar too. This will be the focus of next lecture.



So, because productions have need to be exhaustive PDC which is production coverage, subsumes TSC. If we cover every production we cover every terminal. This is what I told you, derivation coverage is impractical and infeasible also we do not need them. These coverage criteria consider generating strings that are members of the language. When we do mutation testing we will see how do generate strings that are not members of the language. So, that will be the focus of the next lecture.

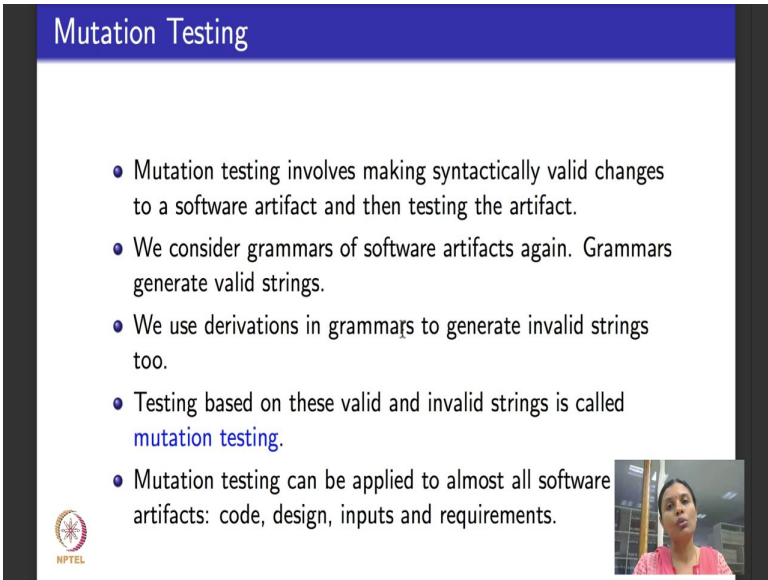
Thank you.

Software testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 36
Mutation Testing

Hello, welcome to the second lecture of week 8. So, what are we going to do today? Last time I told you that mutation testing means testing for the syntax of a program, syntax of a program deals with regular expressions and grammars? So, I gave you a very brief quick introduction to regular expressions and context free grammars and this time what I am going to do is I am going to introduce you to mutation testing.

(Refer Slide Time: 00:38)



The slide is titled "Mutation Testing" in a blue header. It contains a list of five bullet points explaining the concept. The NPTEL logo is in the bottom left corner, and a small video inset of the professor is in the bottom right corner.

- Mutation testing involves making syntactically valid changes to a software artifact and then testing the artifact.
- We consider grammars of software artifacts again. Grammars generate valid strings.
- We use derivations in grammars to generate invalid strings too.
- Testing based on these valid and invalid strings is called **mutation testing**.
- Mutation testing can be applied to almost all software artifacts: code, design, inputs and requirements.

Next week we will see how to apply mutation testing to source code. So, what is mutation testing involve? The term mutation in the in biology or generically means you make a change. So, in biology you make changes cells undergo mutation they undergo changes. When we apply mutation testing in the context of software testing, we say a software artifact undergoes the change.

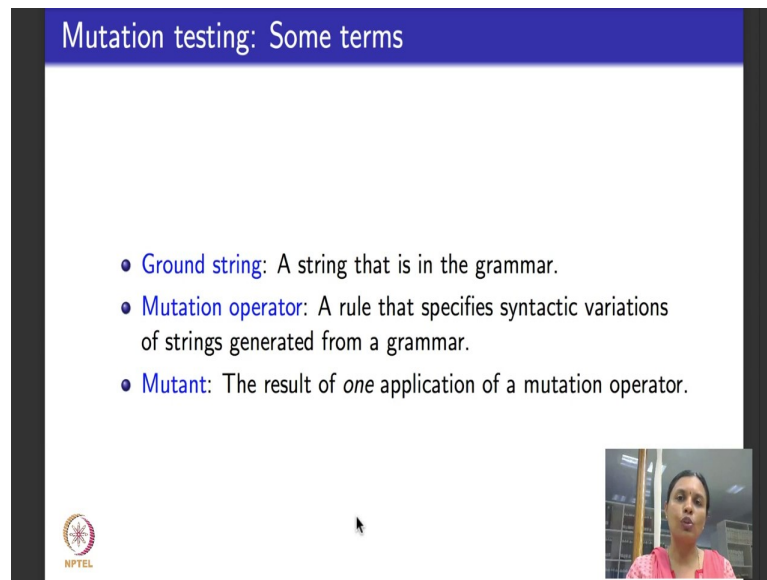
So, the software artifact that could be mutated or changed could be a program, could be an input, could be a design document, it could be one of the several different things. So, mutation testing means you make changes, most of the time the changes are syntactically valid, I will explain to you what that is to a software artifact and then test the artifact.

When I say syntactically valid change I mean the following. So, you consider the software artifact we have be piece of source code or a program now I make a change to see how the changed program behaves with reference to the original program. So, for me to be able to test the changed program the first thing that I must do is to be able to compile the changed program.

So, the changed program must be syntactically valid, it should be a compilable program. So, while doing mutation testing we mutate by making changes. How are the changes defined? The changes are defined by using mutation operators and where do the mutation operators come from they come from grammars of software artifacts. Grammars typically generate valid strings everything to the generate is syntactically valid as per the production rules of a grammar. We use derivations in grammars to generate valid strings. And sometimes we use derivations to generate invalid strings also. When do we generate invalid strings in the grammar? We generate invalid strings only when we consider mutation as being applied to the inputs of a program. When we mutate a program itself then we expect the program to be valid because it needs to be compilable.


But sometimes we want to know how a piece of program, given program, behaves on inputs that are invalid. How does it handle invalid input? So, when we do that we mutates to produce invalid string from the grammar corresponding to the input domain. So, testing based on generating these valid strings and invalid strings for different artifacts is what is called mutation testing mutation testing as I told you can be applied to program source code, it can be applied to design integration, it can be applied to input space and it can be applied to change the requirements themselves.


(Refer Slide Time: 03:08)



Mutation testing: Some terms

- **Ground string:** A string that is in the grammar.
- **Mutation operator:** A rule that specifies syntactic variations of strings generated from a grammar.
- **Mutant:** The result of *one* application of a mutation operator.





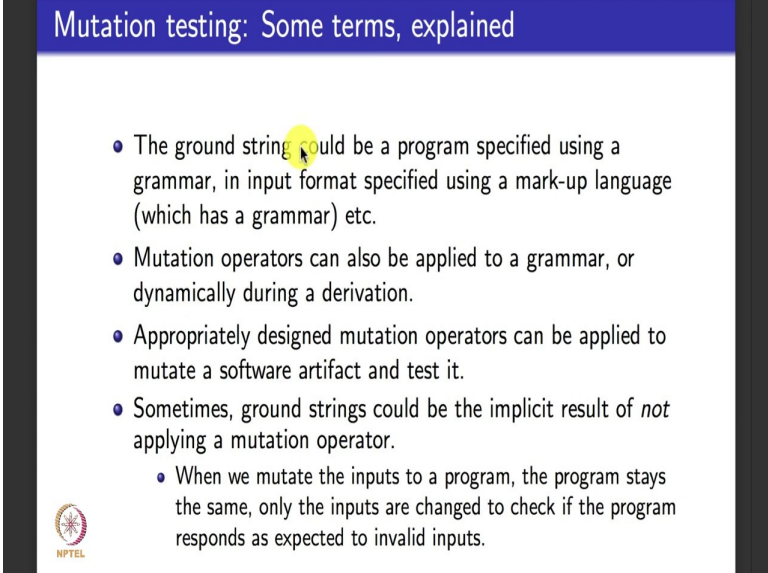
So, here are some terms that we will use throughout our lectures in mutation testing. So, I have a particular software artifact that I am trying to mutate or make a change in, that software artifact is what is called a ground string. So, ground string is a software artifact that I want to mutate. In other words it is the string in the grammar of the corresponding software artifact, it is a string in the underlying grammar, and how do I mutate? I mutate by making what is called a mutation operator. What is a mutation operator? A mutation operator is a production rule or a rule that specifies syntactic variants of strings that can be generated from the grammar. Every string generated by grammar as we saw in the last lecture comes by applying a sequence of rules one after the other till we get a string only of terminals.

At some point during the sequence of derivations that I apply, I decide to apply another rule instead of the original rule that was used in the string and such a rule that I apply instead of the original rule is what is called a mutation operator. So, I begin with the ground string, I apply a mutation operator, after applying mutation operator I have changed a mutated the ground string. The changed or mutated ground string is what is called a mutant. So, a mutant is the result of applying exactly one application of a mutation operator. So, you might ask why exactly one why not more? You can apply more as long as the changes are syntactically valid and you are confident of how the mutated program will behave, but typically in mutation testing it is recommended that

you apply mutation operators one at a time. So, we will discuss this, we will revisit this question once again very soon in this lecture.


But just to move on, before we move on I would like to summarize, let us say you testing a java program using mutation testing, the original Java program that you begin with is called a ground string. You take the Java program apply one mutation operator, that is one change to the underlying grammar from which the Java program comes by applying some rule instead of the other rule the change that you make is called mutation operator or applying a mutation operator. The resulting string is another syntactically valid Java program whose behavior you want to test against the original Java program.

(Refer Slide Time: 05:36)



Mutation testing: Some terms, explained

- The ground string could be a program specified using a grammar, in input format specified using a mark-up language (which has a grammar) etc.
- Mutation operators can also be applied to a grammar, or dynamically during a derivation.
- Appropriately designed mutation operators can be applied to mutate a software artifact and test it.
- Sometimes, ground strings could be the implicit result of *not* applying a mutation operator.
 - When we mutate the inputs to a program, the program stays the same, only the inputs are changed to check if the program responds as expected to invalid inputs.

 NPTEL

So, the changed the Java program is what is called a mutant. So, the ground string could be a program as I told you could be a Java program, it could be specified using the grammar of Java, it could also be an input format specified using a markup language.

Several different things take XML format, several different software programs take XML format corresponding to an entity as their input format. Sometimes I want to be able to test the software artifact by changing the input to a syntactically invalid input or another valid input and see how the program behaves under these invalid inputs. When I do that I used= the grammar to generate implicit invalid inputs also. So, when I do that I get invalid inputs and in that case ground strings result as those that arise by not applying a mutation operators. For example, as I told you when we mutate the inputs to a program;

program remains the same it is still the ground string, but the mutation operator is applied on the inputs to a program the program and the resulting mutant are the same, but the mutated input is what is changed and in thus this cases it can be invalid also.

(Refer Slide Time: 06:48)


Example of mutant

Consider the grammar that we saw in the last lecture:


```

stream := action*
action := actG | actB
actG   := "G" s n
actB   := "B" t n
s      := digit1-3
t      := digit1-3
n      := digit2 · digit2 · digit2
digit  := 0|1|2|3|5|6|7|8|9

```



- Strings generated by the grammar: G 17 08.01.90, B 13 06.27.94
- Two valid mutants: B 17 08.01.90, G 43 08.01.90
- Two invalid mutants: 12 17 08.01.90, G 23 08.01



So, here is a simple example of how to apply mutation at a basic grammar level. Next lecture I will show you how to apply mutation to source code by taking concrete examples of programs. If you remember we saw this grammar in the last lecture and these was some valid strings that we were generated by grammar. This grammar used notations that were a combination of regular expression notations and grammar notations and this vertical bar is to be read as 2 rules: this rule and then this rule. So, here are some examples of strings generated by grammar. This was generated by this grammar G 1708.01.90 and B 13 06 27 94. Here are 2 examples of valid mutants.

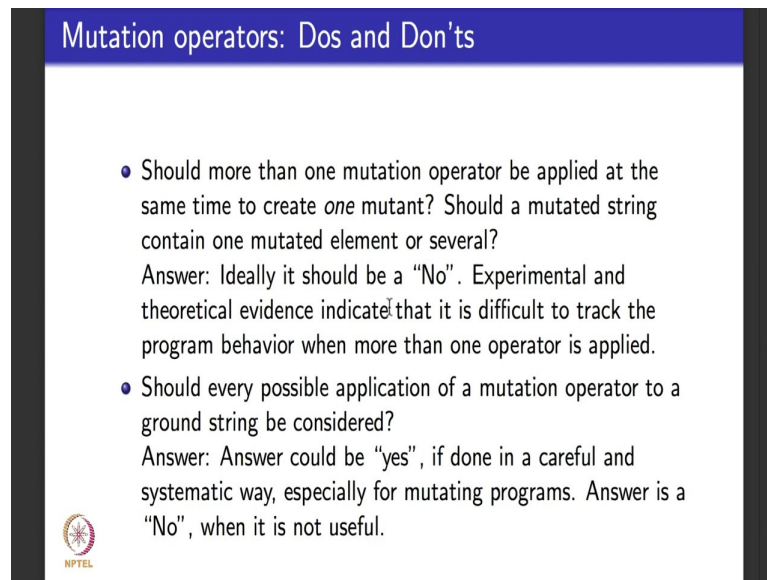
How did I get this? How did I get the first mutant? I took this string this string had a derivation in the grammar at some point this G was derived. In fact, right in the beginning G will be the first terminal string to drive after the third or fourth step in the derivation when this production rule is applied. Instead of now this G whenever this production rule was applied, I picked up the same derivation which gave this at that point instead of deriving G, I applied the 4 production rule and got B instead. So that is the only change. So, at some point in the derivation of this string I decided to take out one

step in the derivation and replace it with a different production rule. In particular whatever was giving me G I replaced it with B. So, this is a valid mutant.

So, here is another example. I take the same string G 17 08 01 90. At some point this number 17 was generated because this rule was applied S goes to digit 1 2 3. It picks up randomly a digit that is between 1 and 3 length long and applies it and here the digits that was picked up was 1 and 7 and it was of 2 digits long. Instead of picking up 1 and 7 at pick up 4 and 3. So, from this I get this valid mutant. Here are examples of 2 invalid mutants from the string, the first invalid mutant says you take this string G 17 08 01 90; instead of G you get 12. Why is this an invalid mutant? If you see this grammar you will realize that no there is no way that this particular string can be generated you can never general 12 17 08 01 090. The way this grammar structured every word of this form we will begin with the G or a B because it begins with the 12 and it cannot be generated by the rules or the grammar we call it an invalid mutant. Maybe these are inputs to a program and you want to check if the program throws in an error saying this input format is invalid.

Here is another example of an invalid mutant. This is the same thing G 23 08 01 and the last 2 digits of this date like entity is missing. Here is another, this is an example of invalid mutant because there is no way I can generate this string from this grammar again because when I reach this stage where I apply this GSN or BTN and I finished applying for S, now I have to apply for n at 1. In 1 shot I generate this whole thing I generate digit 2 dot digit 2 dot digit 2. So, this one if you see does not have the second dot and the third 2 digits. So, it can never be generated in this grammar.

(Refer Slide Time: 10:12)



The slide has a blue header with the text "Mutation operators: Dos and Don'ts". Below the header, there are two bullet points, each followed by an answer. The first bullet point asks if more than one mutation operator should be applied at the same time to create one mutant, and the answer is "No", citing experimental and theoretical evidence. The second bullet point asks if every possible application of a mutation operator to a ground string should be considered, and the answer is "yes" if done carefully, but "No" if not useful. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.

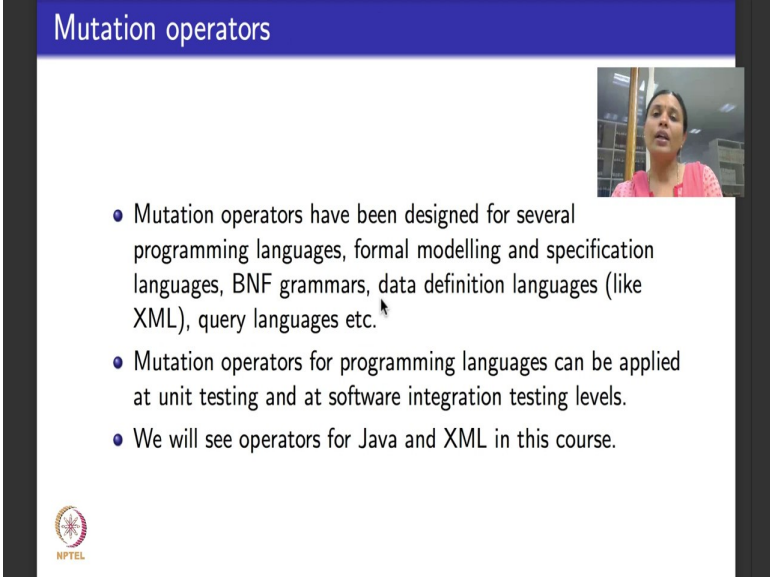
- Should more than one mutation operator be applied at the same time to create *one* mutant? Should a mutated string contain one mutated element or several?
Answer: Ideally it should be a "No". Experimental and theoretical evidence indicate that it is difficult to track the program behavior when more than one operator is applied.
- Should every possible application of a mutation operator to a ground string be considered?
Answer: Answer could be "yes", if done in a careful and systematic way, especially for mutating programs. Answer is a "No", when it is not useful.

So, that is why it is an invalid mutation. So, here are some do's and do not's as far as mutation operators are concerned. The first question that we discussed a little while ago in this lecture was, should I apply exactly one mutation operator to get a muted mutant or should I apply more than one mutation operator to get a mutant?

So, that is the first question. So, should it contain just one mutation operator or it can contain several mutation operator. The ideal recommendation that people use a mutation testing is do not do more than one. Work with exactly one mutant, it is good enough and after analyzing that you work with the next mutant. Experimental and theoretical evidence indicate that it is usually very difficult to track the program behavior when more than one operator is applied. Because you do know whether the change in the mutated program is because of which operator that I apply. Even if you apply to you do not know with the change might be difficult for large programs to isolate and say that this changes because of the first operator, this behavior changes because of the second operator it is very difficult to do that. So, always the voice thing to do is to apply one mutation operator at a time. The second question that you might want to ask you should every possible application of a mutation operator to a ground string be considered? In the sense that I have a grammar of a programming let us say language, a grammar usually fairly exhaustive and if you try to generate mutation operators based on the grammar you will get lots and lots of operators, hundreds, sometimes even thousands of operators.

So, now for a given program they could be several different ways of mutating it. Should I consider every possible mutation and test that is like exhaustive testing with reference to mutation. The obvious answer is no, but sometimes for small programs people say yes also. So, yes if you can do it carefully and in a systematic way know if you do not know that it is not useful. Typically it is not useful to do exhaustive mutation testing.

(Refer Slide Time: 12:14)



The slide is titled "Mutation operators" in a blue header. It contains a list of three bullet points. In the top right corner, there is a small video inset showing a woman with dark hair wearing a pink top, speaking. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.

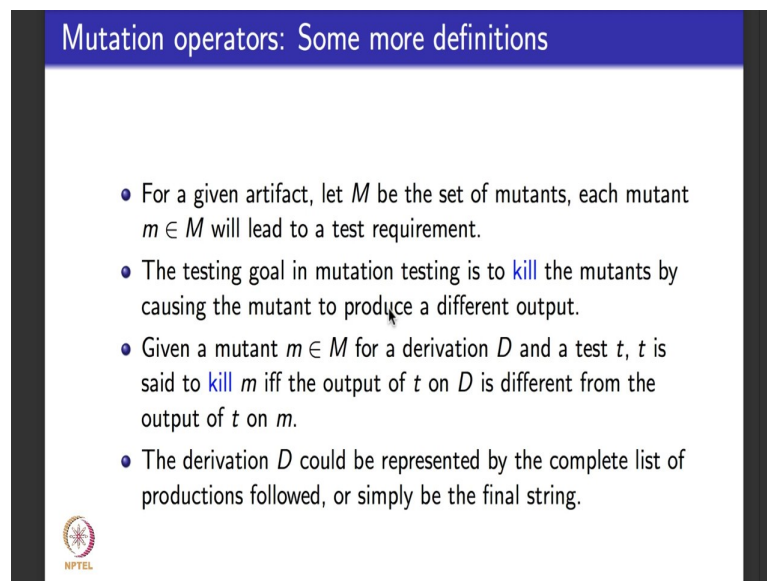
- Mutation operators have been designed for several programming languages, formal modelling and specification languages, BNF grammars, data definition languages (like XML), query languages etc.
- Mutation operators for programming languages can be applied at unit testing and at software integration testing levels.
- We will see operators for Java and XML in this course.

So, the answer is a no. Now, what are the mutation operators? Mutation operators take the grammar and make one change with reference to the grammar that is what I told you. Mutation operators have been designed for several different programming languages. We will see one for java in the next couple of lectures they have been designed for formal modelling and specification languages especially for model checkers like new SMV and SMV, they have been defined for several different grammars and Backus Naur form. I keep using this term be enough I did not really introduce you to you the term be enough we just saw context free grammars.

But there almost always given in Backus Naur form the fact that it comes in this normal form may not be very important for our lectures. So, I skipped introducing the BNF part of the context free grammars when we did context free grammars. We do not really need, but it is important to know that they do not come in any format they always come in BNF format. Mutation operators are also available for data definition languages like XML, they are available for several different query languages SQL and so on. Mutation


operators for programming languages, which are the phases they can be applied in ? They can be applied while doing unit testing and valuing integration testing there are integration level mutation operators that are also available. So, what we will see from the next lecture on words is we will see some mutation operators for java for good number of lectures and when I end mutation testing towards the end we will see mutation operators for XML also.

(Refer Slide Time: 13:44)



Mutation operators: Some more definitions

- For a given artifact, let M be the set of mutants, each mutant $m \in M$ will lead to a test requirement.
- The testing goal in mutation testing is to **kill** the mutants by causing the mutant to produce a different output.
- Given a mutant $m \in M$ for a derivation D and a test t , t is said to **kill** m iff the output of t on D is different from the output of t on m .
- The derivation D could be represented by the complete list of productions followed, or simply be the final string.

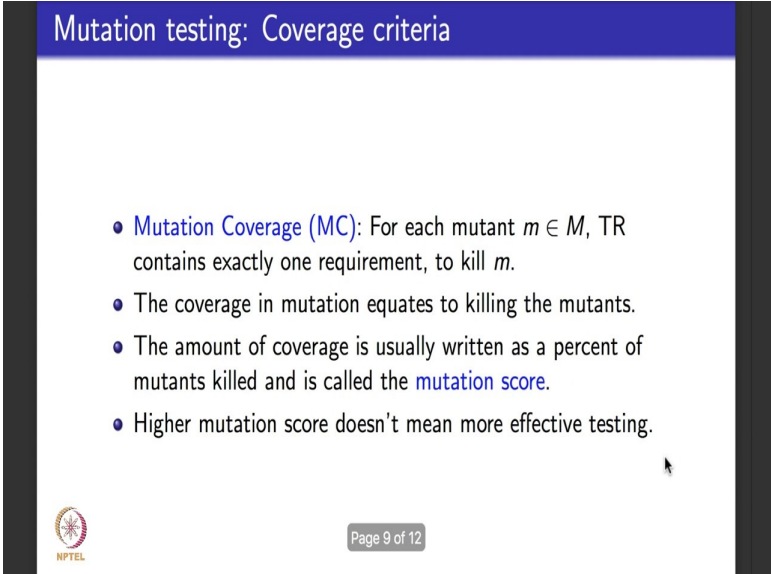


Some more definitions related to mutation testing before we move on. For a given artifact like source code or requirements or design let m be the set of total number of mutants. Each mutant m in m we will lead to a different test requirement. Testing goal in mutation testing is to be able to kill the mutants. What does it mean to kill the mutant? It means the following you take the original program called ground string apply one mutation operator get a mutant or a mutated program. Now take a test case, run the original program on that test case, take the mutated program, run the same test case on the mutated problem. If these 2 programs, the original program and mutated program, produce outputs that are different for the same test case then you say the test case has killed the mutant right. So, given a mutant m for a derivation d and test t , the test t is set to kill the mutant m if and only if the output of t on d is different from the output of t on m . I hope that is clear right. Take a program, apply one mutation operator, get a mutated program. Pick a test case, run it on the original program, run it on the mutated program.

If the 2 programs produce different outputs then you say the test case is killed my mutant.

Is it definition is very important because the core of mutation testing and this lies and understanding or designing test cases for killing mutants. So, the derivation d could be represented by the complete list of productions or simply by the direct result and final string which could even be the program.

(Refer Slide Time: 15:24).



The slide has a blue header with the text "Mutation testing: Coverage criteria". Below the header, there is a white area containing a bulleted list of four points. At the bottom left of the slide is the NPTEL logo, and at the bottom center is a grey box with the text "Page 9 of 12".

- **Mutation Coverage (MC):** For each mutant $m \in M$, TR contains exactly one requirement, to kill m .
- The coverage in mutation equates to killing the mutants.
- The amount of coverage is usually written as a percent of mutants killed and is called the **mutation score**.
- Higher mutation score doesn't mean more effective testing.

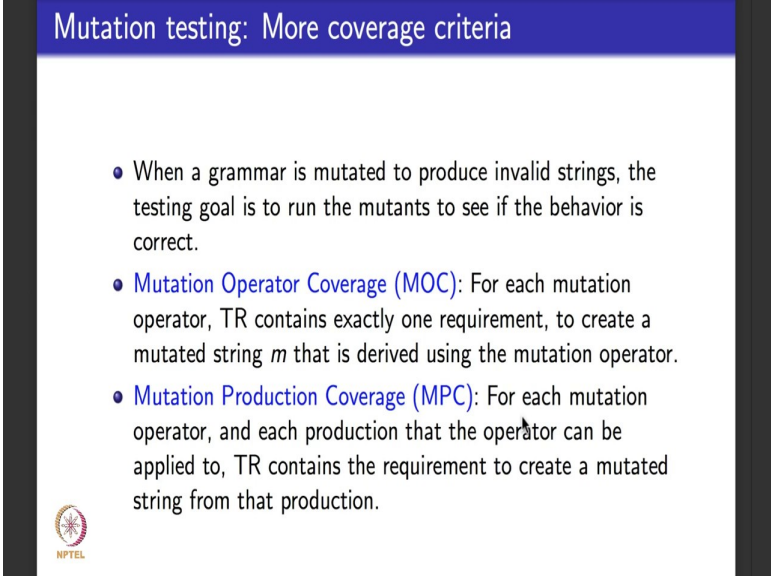
So, here are some coverage criteria for mutation testing. The first coverage criteria that we define is what is called mutation coverage. So, for each mutant that you apply to the original ground string, for each mutant m in M , test requirement or TR contains exactly one requirement, you kill that mutant. So, when I make a change to a ground string and get a mutant I should be able to write a test case to kill the mutant. So, if I am able to do this for every mutant then I have achieved what is called mutation coverage. The amount of coverage that you achieve in mutation testing is usually written as the percent of mutants killed. Like for example, suppose I have a small Java program and it happens to be the case that there are 15 different mutants that I can get from this Java program which means what I can make 15 different kind of changes one at a time to get 15 different other Java programs.

Let us say out of these I manage to write test cases that kill 15 of these mutations. Let us say the test cases that I right I am able to kill only let us say 10 of these 15 different

mutations that I have done. The remaining 5 mutants of this given Java program are such that no matter what kind of test case it is it can never kill in the sense that the behavior of the original program will be the same as the behavior of the mutated program. If that is the case if you manage to kill only 10 out of 15 mutants then the mutation score says that you have killed only 10 out of 15 mutants which means you have roughly killed about 75 percent of the mutants that you created. So, the amount of coverage is usually means the amount of killing of the mutants that you can do, it is usually written as a percent of mutants killed and is called mutation score.

Suppose for all the 15 different mutants that you wrote you managed to write test cases that killed all the 15 different mutants that does not mean that you have tested in a more effective way. It is in fact, a myth to belief that the more mutants that I write and kill the more effective that I have tested the program. These 2 do not really correlate to each other. Another kind of coverage criteria over mutation testing or mutation operator coverage and mutation production coverage.

(Refer Slide Time: 17:30)



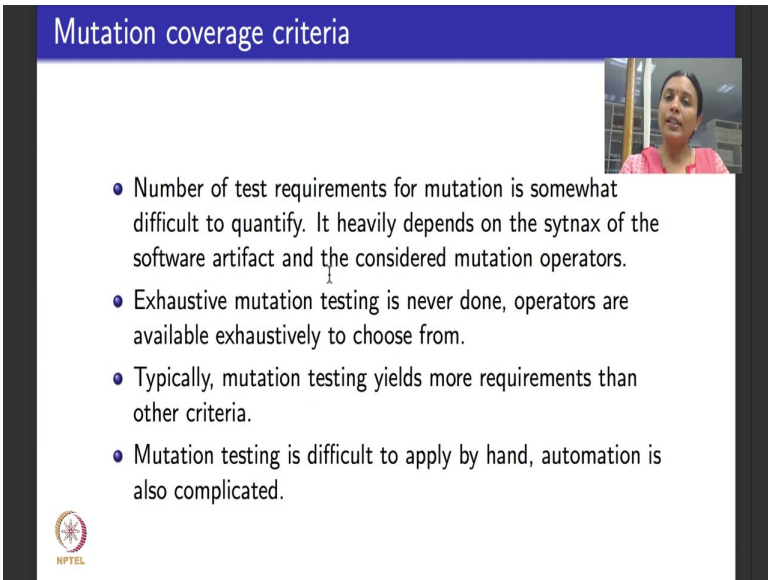
The slide has a blue header with the text "Mutation testing: More coverage criteria". Below the header, there are three bullet points. The first bullet point discusses testing mutants for correct behavior. The second bullet point defines Mutation Operator Coverage (MOC). The third bullet point defines Mutation Production Coverage (MPC). In the bottom left corner, there is a circular logo with a star and the text "NPTEL" below it.

- When a grammar is mutated to produce invalid strings, the testing goal is to run the mutants to see if the behavior is correct.
- **Mutation Operator Coverage (MOC):** For each mutation operator, TR contains exactly one requirement, to create a mutated string m that is derived using the mutation operator.
- **Mutation Production Coverage (MPC):** For each mutation operator, and each production that the operator can be applied to, TR contains the requirement to create a mutated string from that production.

So, when a grammar is mutated to produce invalid strings, like we do for mutating inputs, the testing goal is to run the mutants to see if the behavior is correct. So, in which case, mutation operate operator coverage abbreviated as MOC says for each mutation operator TR contains exactly one requirement, to create a mutated string m that is derived using that particular operator.

So, suppose at some point and time there was an operator that said you use less than or equal to and you are able to create a mutation where instead of less than or equal to you use greater than or equal to. So, this is called one mutation operator. So, I rate a test case to cover this mutation operator. Similarly I can define what is called mutation production coverage. What does that say? For each mutation operator and each production the operator is applied to, test requirement contains a requirement to create a mutated string by using that production. So, the production rule that changed the greater than or equal to 2 less than or equal to use the rule make the change and write a test case to kill that mutant, resulting mutant program.

(Refer Slide Time: 18:44)



The slide is titled "Mutation coverage criteria" in a blue header. It contains a list of four bullet points. In the top right corner, there is a small video inset showing a woman speaking. In the bottom left corner, there is a circular logo with the text "NPTEL" below it.

- Number of test requirements for mutation is somewhat difficult to quantify. It heavily depends on the syntax of the software artifact and the considered mutation operators.
- Exhaustive mutation testing is never done, operators are available exhaustively to choose from.
- Typically, mutation testing yields more requirements than other criteria.
- Mutation testing is difficult to apply by hand, automation is also complicated.

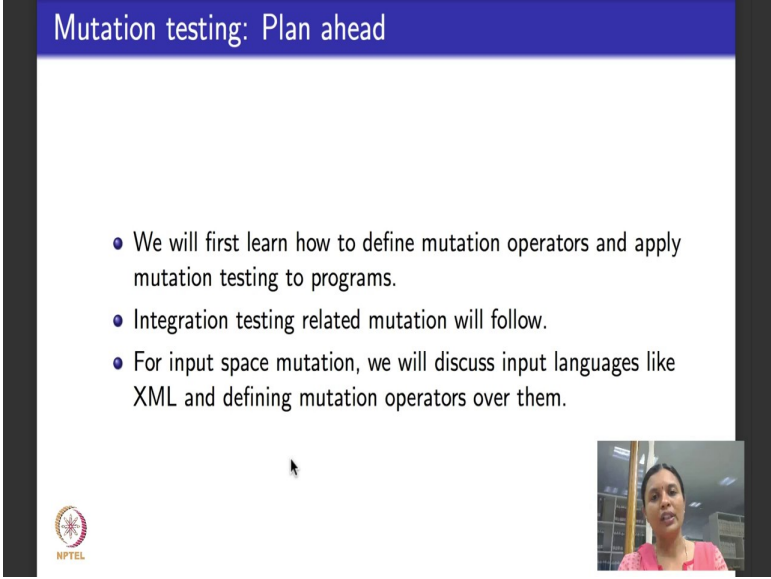
So, these are the 3 mutation coverage criteria. In fact, we will see them for programs and see how they apply. The number of test requirements that I need to achieve mutation coverage criteria is somewhat difficult to quantify.

It basically heavily depends on the syntax of the underlying software artifact or the program and what are the mutation operators that I have considered. Typically as I told you exhaustive mutation testing is never done, but operators are available as a fairly large exhaustive sets to choose from because you never know which operator we will come to use and it is a fairly routine mechanical programming task to generate mutation operators given the grammar of a software artifact. So, even though an exhaustive set of

mutation operators are available, mutation testing is never exhaustively done by applying every possible operator that is available, people usually do not do that.

Typically mutation testing is supposed to subsume a lot of other coverage criteria. We will see precisely a comparison to several other coverage criteria that we saw and how mutation testing subsumes which of those. Mutation testing is very difficult to apply by hand because it involves grammars and there are mutation testing tools available. Towards the end I will point you to links of these kind of tools. In fact, mutation testing is very difficult, it is difficult to apply by hand, it is difficult to automate also. Mutation operators are difficult to automate, but the per say testing process, after you have created the mutated program designing a test case to kill the mutant, there automation is difficult and you need human intervention and domain knowledge. In which operator to apply, they are also you need human intervention and domain knowledge.

(Refer Slide Time: 20:22)



The slide has a blue header with the text "Mutation testing: Plan ahead". Below the header, there is a white rectangular area containing a bulleted list of three items. In the bottom right corner of the slide, there is a small video inset showing a woman speaking. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.

- We will first learn how to define mutation operators and apply mutation testing to programs.
- Integration testing related mutation will follow.
- For input space mutation, we will discuss input languages like XML and defining mutation operators over them.

So, in the next lecture what are we going to see we learn how to define mutation operators how to apply mutation testing to programs or source code to begin with. Then we will apply mutation testing to do design integration. Finally, we will apply mutation testing to a markup language like XML and understand how mutation testing applies to input space to create invalid inputs and how the programs react to invalid inputs. So, that is the plan for mutation testing.

Thank you.



Software testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 37
Mutation Testing for programs

Hello there, welcome to the third lecture of week 8. So, we are going to this lecture see how mutation testing applies through programs. What I am going to tell you today is explain in detail the terms that we saw in the last class. Ground string, mutation operator mutant, killing a mutant, will re understand all these terms, but by applying it to a particular program and then we will do a couple of examples explaining how to write test cases that will kill the mutants for this programs.

(Refer Slide Time: 00:46)

Mutation testing for software artifacts: An overview				
	For programs	Integration	Specifications	Input space
BNF grammar	Programming languages	–	Algebraic specifications	Input languages like XML
Summary	Compilers			Input space testing
Mutation	Programs	Programs	FSMs	Input languages like XML
Summary	Mutates programs	Tests integration	Model checking	Error checking



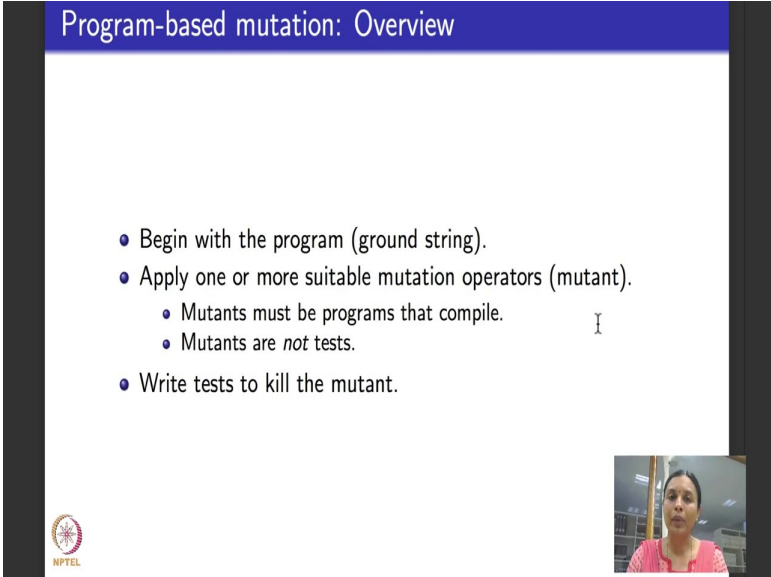
So, here is an overview of the rest of how the courses is going to go for mutation testing. Till now we saw grammars, I did not introduce you to BNF, but I introduced you to a context free grammar as it occurs in the syntaxes several programming languages and when I apply grammar based testing through programming languages, I usually test compilers. Compiler testing is a very involved area we want deal with it in the course.

No known application of direct grammar based testing is available for design integration. Grammar based testing is available for algebraic specifications. Again because this we will involve deep algebraic specification knowledge I am going to skip this part of the

course, but we will apply grammar based testing to do input formats like XML towards the end of this module of mutation testing, we will see this part. We also saw mutation testing in the previous lecture. Mutation testing again applies to all the software artifacts, it applies to program source code of programs where it mutates programs we will see that today and in the next lecture. Mutation testing can be applied for design integration it mutates operators that led to the test basically how one module calls another module and all the integration operators that a particular programming language has to offer. Mutation testing can be applied to specifications especially to finite state machines. There are mutation operators is available for model checkers like SMV, NuSMV but because this course is not on modeled checking I will skip this part, but we will see mutation has its apply to input languages like XML.

So, what are we going to do today's lecture we will begin with program based mutation continue into the next lecture after that I will tell you program based mutation, but for integration testing and then we will see how mutation testing compares for producing invalid inputs by applying it to languages like XML.

(Refer Slide Time: 02:49)



The slide is titled "Program-based mutation: Overview" in a blue header. It contains a bulleted list of three main steps:

- Begin with the program (ground string).
- Apply one or more suitable mutation operators (mutant).
 - Mutants must be programs that compile.
 - Mutants are *not* tests.
- Write tests to kill the mutant.

In the bottom right corner, there is a small video inset showing a woman with dark hair wearing a pink top, who appears to be the presenter. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.

So, that is the plan for the rest of the modules on mutation testing. So, of program based mutation which we are going to begin in this lecture what do we do? We basically begin with the given program. Please remember that in mutation testing the given program that you begin with is called the ground string. Then I apply one or more suitable mutation

operators to get a mutant or a mutated operator, mutated program. Typically its exactly one mutant that I apply. Remember that mutants must be valid programs that must compile for programs for sure and mutants are not test cases. We have to write test cases to kill a mutant. What was the definition of killing a mutant that we saw? Killing a mutant means that the behavior of the test case of the original ground string program is different from the behavior for the same test case in the mutated program.


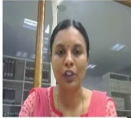
(Refer Slide Time: 03:39)

A simple example of mutation, contd.

One mutation of the Min method:

```
int Min(int A, int B)
{
    int minVal;
    minVal = A;
    if (B<A)
    {
        minVal = B;
    }
    return(minVal);
}
```

```
int Min(int A, int B)
{
    int minVal;
    minVal = A;
    Δ1 minVal = B;
    if (B<A)
    {
        minVal = B;
    }
    return(minVal);
}
```



Here is a simple example. What is this Java, what is this program to, it could be Java it could be C whatever it is the if I have just given a tiny little code segment does not matter which programming language it is. So, here is his method called min it takes 2 arguments A and B which are both integers and then it computes the minimum of A and B. That is what the job is supposed to do, which is the less amongst A and B. So, it initializes a local variable called min value where it stores the value of the minimum it says let min value be A to start with. If B happens to be less than a then you change the min value to be otherwise let min value continue to be whatever it is A or B it returns the min value. Simple enough right? Suppose we want to test this program using mutation test what do we do. So, here is the original method given on the left hand side. What I have done it is I want to check whether this program correctly computes the minimum value.

So, I will say is this code correct I assigning minimum value to be A and then say if B is less than A then you reassigned minimum value to be B. Is that correct? What will happen if I use the same code, but instead of assigning minimum value to A, I assign minimum value as B. So, that is called one mutation. So, I take the same program which is given on the left hand side and then I consider the same program and I make one mutation which is given here. How will you read this as? Delta one this triangle or delta one read it has one mutation applied to this program on the left which is the ground string. What is the mutation? Instead of this statement, `min val A` you remove that statement and replace it with `min val B`. In other words, assuming that this program came from a grammar at some point in the grammar the string the statement has a string would have got generated `min val A`.

At the same time instead of generating a substitute A with B. So, on the right hand side how do I read this? This is the same copy of the program which is the ground string on the left hand. So, the ground string is on the left hand side, the mutant or the mutated program is on the right hand side. What is the difference between the ground string and the mutant? That is this one statement. The statement `min val equal to A` is not there. Even though I have returned it here for the sake of completeness in the muted program it is not there instead of that this statement is there `min val equal to B`. So, what is the mutation or the change that I have done? I have taken this statement in the ground string changed this A with B. That is how you read the program on the left hand side. Is that clear? Instead, in fact, to this is just one change you could make several other changes.


(Refer Slide Time: 06:21)

A simple example of mutation, contd.

Six different mutations of the Min method. Results in six different programs, each with one mutation.

```
int Min(int A, int B)
{
    int minVal;
    minVal = A;
    if (B<A)
    {
        minVal = B;
    }
    return(minVal);
}
```

```
int Min(int A, int B)
{
    int minVal;
    minVal = A;
    Δ1 minVal = B;
    if (B<A)
    Δ2 if (B>A)
    Δ3 if (B<minVal)
    {
        minVal = B;
    }
    Δ4 Bomb();
    Δ5 minVal = A;
    Δ6 minVal = failOnZero(B);
    return(minVal);
}
```



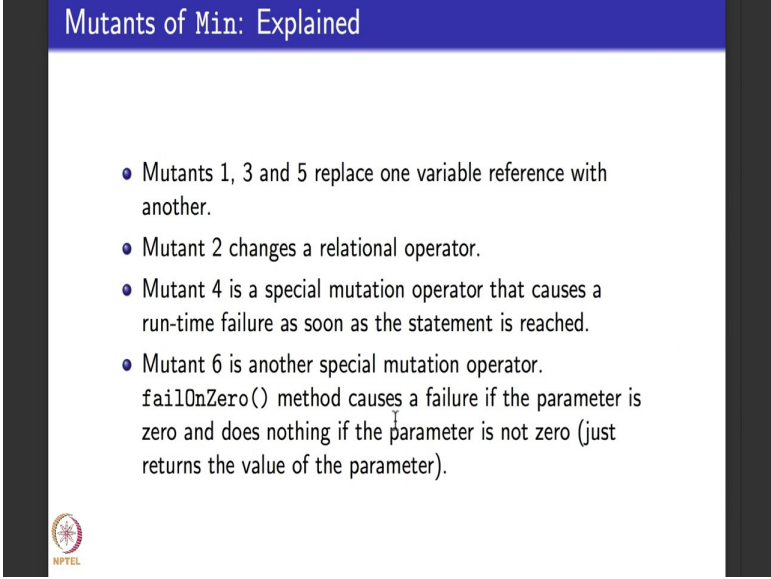
So, what I have done here is a given you examples of 6 different mutants for the same min method. How do I read this? 6 different mutants the one on the left hand side is the ground string or the program that is being tested for mutation testing. I can create 6 different mutants how do I create 6 different mutants, they are labeled here: delta 1, delta 2, delta 3, delta 4, delta 5 and delta 6. How do these come into picture? delta 1 means take out this statement min val equal to A replace it with min val equal to B. All other parts the program remain the same. What is delta 2 mean ? Take out the statement if B less if B less than A, instead of that, keep this statement if B greater than A. So, that gives me a second mutated program from the ground string on the left.

Similarly, the third mutated program says take out this if statement B less than A, instead of that keep this B less than min val. And what do these mean? These mean that you insert a new statement called bomb instead of min val equal to B and I tell you what the statement bomb is. It reads the little funny, but it make sense. The fifth mutation says instead of min val equal to B. What will happen if you do min val equal to A, that is the fifth mutation. Sixth mutation says instead of doing min val equal to B you do min val is equal to fail on 0 of B. Fail on 0 is again another special mutation operator for Java, I will tell you in a minute what it is. The rest of the program is same. So, how do I understand this? The program on the left hand side is called the ground string. From the ground string I pick one statement at any point and time and I make a change to that

statement. On the right hand side I have shown you put together in one consolidated program how 6 different changed mutated programs look like.


The first mutated program takes this statement min val equal to A, replaces A with B the rest of the program is the same. The second change is you check the statement B less than a instead change it to B greater than A, that is the second mutation. Third mutation says instead of the statement B less than A change to B less than min val. Forth mutation says instead of min val equal to B call this method called bomb. Fifth mutation says instead of min val equal to B call min val equal to A. Sixth mutation says instead of min val equal to B call min val equal to fail on 0 of B. Please remember even though I have put it inside one program, what I actually I am depicting in the single program are 6 different mutants that you can apply. At any point in time one at a time to the given ground string program on the left to get 6 different mutated programs.

(Refer Slide Time: 09:18)



Mutants of Min: Explained

- Mutants 1, 3 and 5 replace one variable reference with another.
- Mutant 2 changes a relational operator.
- Mutant 4 is a special mutation operator that causes a run-time failure as soon as the statement is reached.
- Mutant 6 is another special mutation operator. `failOnZero()` method causes a failure if the parameter is zero and does nothing if the parameter is not zero (just returns the value of the parameter).



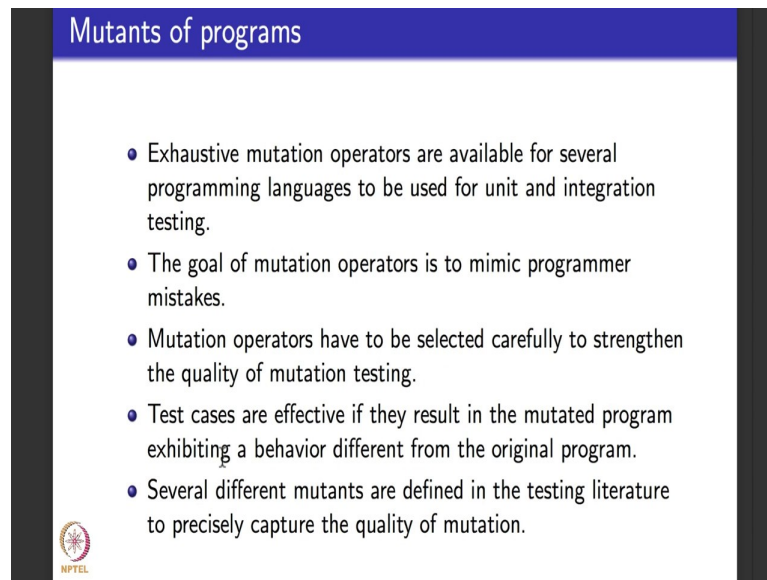
So, this one consolidated thing corresponds to 6 different mutations, please remember that. So, this is what I explained to you. Mutants 1, 3 and 5 are basically called variable reference mutants because they replace one variable reference for the other. Let us go back and see, mutant one replaces A with B, mutant 3 replaces A with min val, mutant 5 replaces B with A. So, it just changes the variable reference. Mutant 2 changes a relational operator. Let us see that. If you go back mutant 2 changes this less than 2 greater than that is what is this mutant 2. Mutation 4, I told you right, that is the bomb

statement which seem related to funny that is a special mutation operator. It is called like that because it causes a runtime failure as soon as the statement is reached. You might wonder what is the point in putting the bomb statement instead of doing this min val equal to B.

Typically there is something called random testing or crash testing a program bomb statement is very useful to see how the program behaves when it randomly crashes. So, by apply this bomb mutation by replacing min val equal to B with that bomb statement I am checking for random crashes in the program. The 6th mutant is another very special mutant operator this mutant fail on 0 what is does is it is a method that causes of failure if it is parameter is 0. If the parameters non 0 it does nothing. Does nothing meaning what? It just returns the value of the original parameter as it was. So, if you go back this was the sixth mutation I took this statement min val is equal to B in the ground string and instead of B I put fail on 0 of B. So, it is like saying what would happen to this program if the value of B was 0. In testing usually during unit testing and debugging, it is recommended the programmers give every variable's value as 0 and test.


So, this fail on 0 mutant is very useful for that. It basically take this ground string program. By changing B to fail on 0 B it checks how the program behaves if the parameter B is 0. If the parameter B is non 0 it will just replace it with the parameter and behave as of the original program would behave, but it will tell you what happens if the parameter is 0.

(Refer Slide Time: 11:38)



Mutants of programs

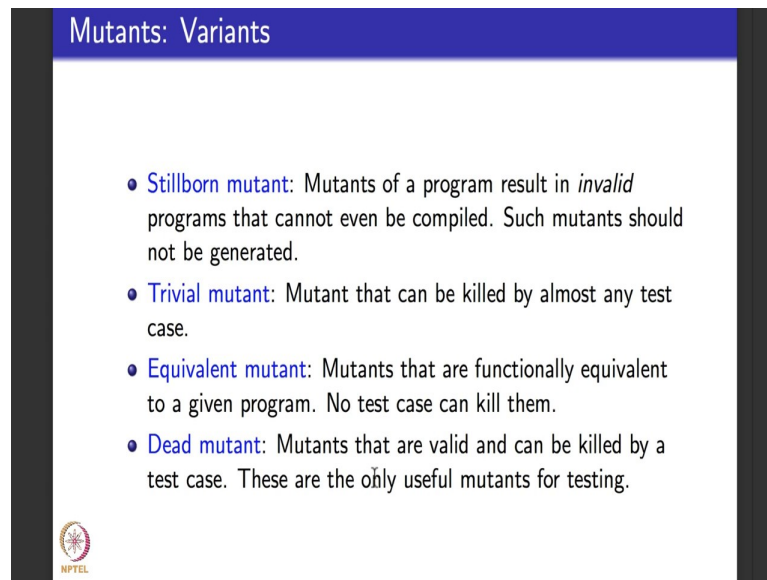
- Exhaustive mutation operators are available for several programming languages to be used for unit and integration testing.
- The goal of mutation operators is to mimic programmer mistakes.
- Mutation operators have to be selected carefully to strengthen the quality of mutation testing.
- Test cases are effective if they result in the mutated program exhibiting a behavior different from the original program.
- Several different mutants are defined in the testing literature to precisely capture the quality of mutation.



So, it is useful for making a variable 0 which is recommended practice in testing and to test a program for that. So, how do I create mutants of programs? As of always told you from the time we began mutation testing, exhaustive list of mutation operators are available for several different programming languages. They can be used for unit testing like we saw in this example of program. They can also be used for integration testing. In the next lecture I will show you exhaustive list of simple mutation operators that can be applied to Java programs and to programs in C.

The goal of mutation operator is to mimic simple programmer mistakes. If you see this example you will understand the statement right. Maybe the programmer made a mistake. This is a correct program, the ground string, may be instead of writing A if we written B then how would the program behave erroneously. Similarly instead of less than suppose the programmer I had written greater than then how would the program behave erroneously? That is what this mutation operators do they make small changes that will reflect typical mistakes that a programmer we will make and then see how the change makes the program behave differently. Mutation operators, which of these operators to apply? We saw 6 for that example the choice is up to us. Sometimes we might want to test for some parameters of the program then you choose mutation operators accordingly. Sometimes you might have audit requirements with say that you test for such requirements then you might have to apply all the recommended mutations, it completely depends on what is the current status of testing.

(Refer Slide Time: 13:25)



The slide is titled "Mutants: Variants" in a blue header. It contains a bulleted list of four types of mutants. At the bottom left of the slide is the NPTEL logo.

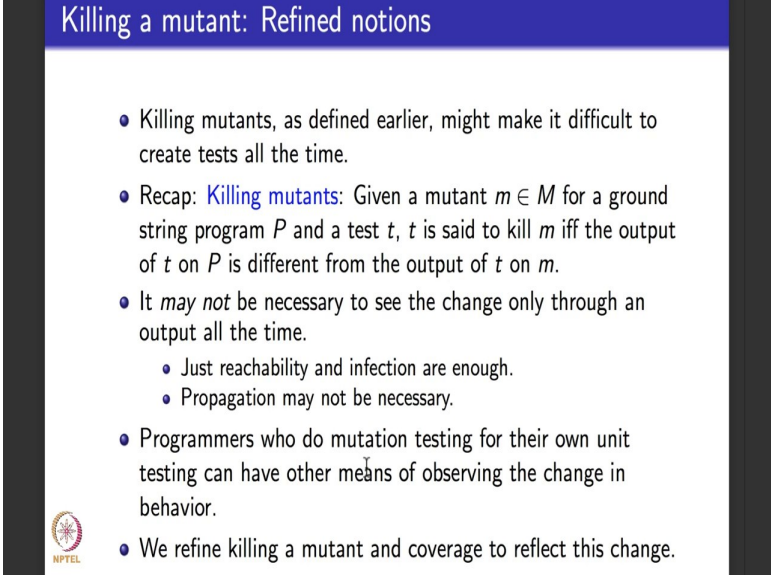
- **Stillborn mutant:** Mutants of a program result in *invalid* programs that cannot even be compiled. Such mutants should not be generated.
- **Trivial mutant:** Mutant that can be killed by almost any test case.
- **Equivalent mutant:** Mutants that are functionally equivalent to a given program. No test case can kill them.
- **Dead mutant:** Mutants that are valid and can be killed by a test case. These are the only useful mutants for testing.

Test cases have to be effective in the sense that they have to make the mutated program come up with the different output than the original ground string programs. And as I told you testing literature is studded with different mutation operators and testing for mutation operators. There are several variants of mutants that one can define. In fact, these terms might be hilarious for that how its lingered on in the mutant mutation testing community. So, you say mutant is still born if the mutants of a program result in invalid programs that cannot even be compiled. As I told you right, when I mutate a program I need another program that is a muted program to be valid, syntactically valid because I need to be able to execute my test case on that. But suppose I mutate in such a way that I get a program that is in tactically invalid and cannot even be compiled then that mutation mutant is what is called stillborn mutant. We do not really need such mutants, they are not useful for testing.

A trivial mutant is a mutant that can be killed by almost any test case. What do we mean by killed by almost any test case? That it is so easy that you do not have to put in much effort to make it different. It is almost easy to apply and it is not useful for testing how the program behavior changes. An equivalent mutant is mutant that is functionally equivalent. In the sense that if you taken the ground string program you made a change the ground string program resulted in a mutant but you no matter what you do you cannot find test case to kill it. In the sense that the ground string program and the mutated program always produce the same output on every possible test case, in which


case the mutant is called an equivalent mutant. There is one more kind of mutant which is called a dead mutant, these are mutants that are valid but they can be killed by almost any test case right. That can be not by almost any test case sorry, but they can be killed by some test case. These are the real mutants that are needed for our testing.

(Refer Slide Time: 15:12)



Killing a mutant: Refined notions

- Killing mutants, as defined earlier, might make it difficult to create tests all the time.
- Recap: **Killing mutants**: Given a mutant $m \in M$ for a ground string program P and a test t , t is said to kill m iff the output of t on P is different from the output of t on m .
- It *may not* be necessary to see the change only through an output all the time.
 - Just reachability and infection are enough.
 - Propagation may not be necessary.
- Programmers who do mutation testing for their own unit testing can have other means of observing the change in behavior.
- We refine killing a mutant and coverage to reflect this change.



So, we saw the notion of killing a mutant what it means? It means that given a program and a test case, I apply mutation to a program get a mutated program. If the test cases such that the behavior of the original program on the test case is different from the behavior of the mutated program on the test case, then you say that the mutant is killed. It may not be necessary to see the change only by observing the output all the time. If you remember we saw these reachability infection propagation and reveal model, RIPR model in the first week of this course. All that I want to know is that the program behavior of the ground string is different from the program behavior of the mutated program. I need not wait for the program to make, reflect the change all the way down the output. Maybe somewhere in between a program I can do a print test to see if the behavior at that point in this statement is actually different, that is all I am interested in.


It might be too much to expect that the if the behavior that is changed at that statement that was mutated in the program is propagates all the way to the output. It might be difficult to do that also. So, sometimes I require that it propagates all the way to the output, sometimes I am happy that the behavior at that mutated statement is different

from the ground string. We have to refine the notion of killing a mutant to understand this.

(Refer Slide Time: 16:36)

Killing a mutant and coverage: Refined notions

- **Strongly killing mutants:** Given a mutant $m \in M$ for a ground string program P and a test t , t is said to **strongly kill** m iff the output of t on P is different from the output of t on m .
- **Strong Mutation Coverage (SMC):** For each $m \in M$, TR contains exactly one requirement to strongly kill m .
- **Weakly killing mutants:** Given a mutant $m \in M$ that modifies a location l in a program P , and a test t , t is said to **weakly kill** m iff the state of the execution of P on t is different from the state of execution of m immediately after l .
- **Weak Mutation Coverage (SMC):** For each $m \in M$, TR contains exactly one requirement to weakly kill m .

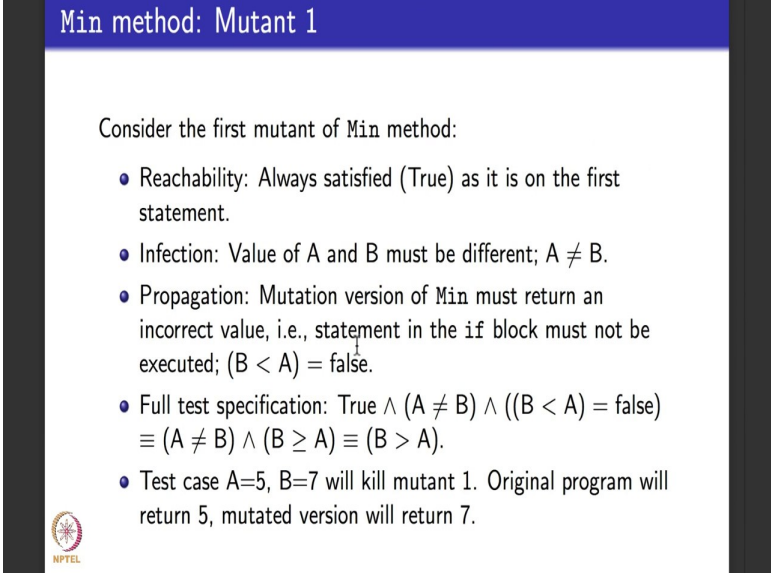


So, we make killing the mutant strongly killing mutants and weakly killing mutants. Strongly killing is what we saw as killing a mutant earlier. Strongly killing mutant means given a mutant, for a ground string program B and the test case t , t said to strongly kill the mutant if the output of the program is different from, the output of the mutated program is different from the output of the ground string program, which means propagation has happened all the way and revealing is also happened, the test case is managed to strongly kill it.

So, I change mutation coverage to strong mutation coverage which says for each mutant test requirement contains exactly one requirement to strongly kill that mutant. A weaker notion of killing a mutant which is what I explained to you know is that I might do a mutation by making one mutation operator to a program, but all that I want to know is that at that statement is there a change in the behavior of the program. I may not expect the change to propagate all the way to the output. That is what weakly killing a mutant captures. So, what is weakly mutant? Given a mutant that modifies a particular location l or a statement in the program and the test case t , t as said to weakly kill them mutant if the state of the program p at, on applying t is different from the state of the mutated program immediately after that statement l at which mutation was applied. These are

clear? So, just as for strong mutation coverage we take mutation coverage and refine it to weak mutation coverage. What is weak mutation coverage say? For each weak mutant M ; TR contains exactly one requirement to weakly kill m .


(Refer Slide Time: 18:23)



Min method: Mutant 1

Consider the first mutant of Min method:

- Reachability: Always satisfied (True) as it is on the first statement.
- Infection: Value of A and B must be different; $A \neq B$.
- Propagation: Mutation version of Min must return an incorrect value, i.e., statement in the if block must not be executed; $(B < A) = \text{false}$.
- Full test specification: $\text{True} \wedge (A \neq B) \wedge ((B < A) = \text{false}) \equiv (A \neq B) \wedge (B \geq A) \equiv (B > A)$.
- Test case $A=5, B=7$ will kill mutant 1. Original program will return 5, mutated version will return 7.



So, we will go back to the min method. So, I will show you the program and the first mutant. So, this was the min method. This is the first mutant that I applied basically take the statement min val equal to A; change it to B. Now I want to understand is this mutant strong can be strongly killed, can be weakly killed, is it a dead mutant, I want to understand what it is. So, what do I do I go ahead and do my reachability infection and propagation condition. Reachability means what I need to be able to reach that statement. If you go back for a minute and see I can always reach the statement because it is write the second statement, so, reachability is true. Propagation means what, I mean infection means what, that this test should be different because this is the only a test based on whether it passes or fails this statement will be executed right. If this passes the result is going to be different if this condition fails the result is going to be different.

So, that is what infection deals with reachability is always satisfied as it is in the first statement. Infection means the value of A must be different from B that is when the if condition will be tested. Propagation means mutated version of minimum must return an incorrect value that is the statement in the if block must not be executed. If it must not be executed means what the condition of the if statement B less than a must be false. So, the

full test specification for reachability infection and propagation says you just AND them, reachability is true, infection is $A \neq B$, propagation is $B \leq \text{false}$. It is simplified this logical predicate true and $A \neq B$ gives me $A \neq B$; $B \leq \text{false}$ is the same as $B \geq A$. So, if and these 2 condition it means $B \geq A$; B is not equal to A and B is greater or equal to A . So, the equal to $A \neq B$ cancel out I just get $B \geq A$.


So, what it means it means that give 2 values which are inputs to the program A and B such that B is greater than A . Then you will be able to kill that mutant. So, some test case like this like for example, A is equal to 5, B is equal to 7 which satisfies this condition B greater than A will kill the first mutant right. Why, because the original program we will correctly return 5 which is the minimum value. Mutated program will return seven because I have change the assignment from B to A and the if condition then didn't pass. Is that clear please.

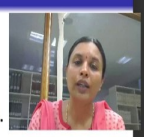
(Refer Slide Time: 20:57)


Min method: Mutant 3

The third mutant of Min method is an equivalent mutant.

- Intuitively, `minVal` and `A` have the same value at that statement in the program, so replacing one with the other has no effect.
- Reachability is true; infection condition is $(B < A) \neq (B < \text{minVal})$.
- It is also true that $(\text{minVal} = A)$ (assertion).
- Simplifying, we get $(A \neq \text{minVal}) \wedge (\text{minVal} = A)$, a contradiction. This means that no value satisfying the conditions exist.







So, now let us look at third mutant we will go back and see what the third mutant is. So, here is the program and here is the third mutant. What is the third mutant do? It changes this if statement in particular it changes the predicate in the if statement if replaces if B less than A it replaces it with if B is less than `min val`. That is instead of doing a directly it replaces it with B is less than `min val`.

So, this is the original program or the ground string. In this program this statement replace like is the mutated program. Lets understand what are reachability infection and propagation conditions for that mutant I believe that it will not have that big a change because if you see just before that in the program the value min val was assigned to A right. So, min val and the variable A have the same value at that statement in the program, when that if statement is executed in the program. So, even normally you should understand that replacing one with the other in the if statement should not have any effect, which means what. Let us look at what how does it reflect on the reachability infection and propagation conditions. So, reachability as always is true because I can always reach that statement. What does infection condition mean? Infection condition is that the if should be test which means B should be less than A and the fact the B is less than A should not B the same as B been less than min val because the mutant replaces A with min val right.


Independently just from the statement before this we know that min val is equal to A, we know that min val is equal to A. So, what we mean by that? We know at because the statement holds if I simplify it what do I get? I will get a is not equal to min val from this part because B is less than A and it is not the same as B is less than being min val. And I also know from this assertion that min val equal to A. What do, if read this independently you see that A is not equal to min val and A is equal to min val. This is the contradiction right. This the contradiction logic means what it means that I cannot find any test case value that will do infection and propagation for this condition, which means this is an example of an equivalent mutant.

(Refer Slide Time: 23:25)

Another example

```
1  boolean isEven(int X)
2  {
3      if(X < 0)
4          X = 0-X;
5      if(float)(X/2) == ((float)X)/2.0
6          return(true);
7      else
8          return(false);
9  }
```

NPTEL





So, to understand reachability infection and propagation well, we will go through another small example. So, here is an example that tests if a number, given number, is Boolean or not. If it is Boolean it returns true otherwise it returns false. So, Boolean is a method I mean, is even is a method that returns a Boolean value, takes an integer as its argument. It first checks if X is less than 0, if X is less than 0 it makes it a positive number then it assigns X to 0 and then divides it by 2. If it divides it by 2 and every number should be divisible by 2, that is what this check condition, if it is divisible by 2 then it returns true.

Otherwise it returns falls this program is simple enough to be clear. Let us look at the reachability infection and propagation for one mutant in the program. What is this mutant? This mutant instead of keeping this original statement as X is equal to 0 minus X replace is with X is equal to 0.

(Refer Slide Time: 24:25)

Another example, contd.

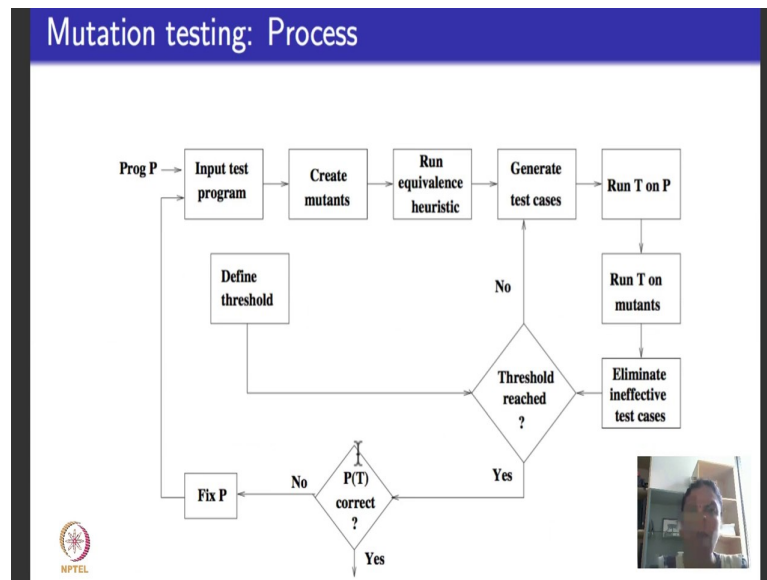
- The mutant in line 4 is an example of weak killing.
 - Reachability: $(X < 0)$.
 - Infection: $(X_T \neq 0)$.
- Consider the test case $X = -6$. Value of X after line 4:
 - In the original program: 6
 - In the mutated program: 0.
- Since 6 and 0 are both even, the decision at line 5 will return true for both the versions, so propagation is not satisfied.
- For strong killing, we need the test case to be an odd, negative integer.



So, again what is reachability for this mutant. Reachability means this if condition should be passed. That is how I get to statement number 4. For this if condition to be pass value of X must be less than 0. So, that is what is returned here X is less than 0 for it to be infected X must not be equal to 0 why if X is equal to 0 then both 4 and the mutated version of 4 we will return the same value, but if X is a number that is not equal to 0 and in fact, less than 0 then mutated version of 4 we will make it 0, but the non mutated the original 4 we will not make it 0. So, infection is that.

So, let us consider a test case of the form X is equal to minus 6 right. So, in the original program the value will be 6. In the muted program the value will be 0. But then if you see both 6 and 0 are even numbers which means both of them will become divisible by 2 at line number 5 in which case the program will return true for both versions, I want the program to return false value for one version. How do I do that ? I have to give odd negative integer. If I give an odd negative integer then here there will be infection and propagation will also happen because the condition in line number 5, will return 0 as even number because of the mutated version of the program whereas, in the original version that it will be still be an odd positive number which is not divisible by 2. So, it will return it as false. So, far reachability infection and propagation which results in strong killing of mutants, I want the test case to be odd negative integer.

(Refer Slide Time: 26:06)




So, here is the summary slide that explains the process of mutation testing. I begin with the program P , my ground string which is my input program to be tested by using mutation. I create one or more mutants for the program. How do I create mutants? I have a standard list of mutation operators which I will be telling you in the next module. Using one or more of those mutation operators, I create different mutants. Please remember that per one mutant I have to use only one mutation operator, it is not recommended that you use more than one mutation operator. Now if the mutant turns out to be equivalent to the original program then there is no point in trying to test it. So, usually there are these things called equivalence checkers which return yes or no for a large number of cases which basically tell you if the mutated program is the same as the original program.


If it is then there is no point. If it is not then I generate test cases and I run the test cases on the program first, then I run the test cases on the mutated program. Because they are not equivalent the output should be different. If the outputs are not different because propagation is not achieved then the test case is called ineffective, I eliminate it. And I will usually have a threshold that is defined independently by let us say a quality auditor or somebody which tells you how many mutants to test the program for. If I have reached my threshold then I finish, exit. If there are any errors that I found then I fix the program and I go back. If I have not reached my threshold then I generate more test cases and repeat this process. So, this is how broad level mutations testing works step by step.

(Refer Slide Time: 27:48)

Mutation testing for source code: Summary




- Mutation testing for source code is one of the most powerful forms of testing source code.
- We need to generate an *effective* set of test cases.
- Next lecture: Mutation operators for typical source code.




So, here is the summary of mutation testing for source code. Its one of the most powerful forms of testing source code. In later this week I will tell you how mutation operators and mutation testing subsume several other coverage criteria that we have seen. It is very powerful for both unit testing and for integration testing. And then the biggest problem is we need to pick up an effective set of mutants that will give us an effective test setup test cases such that if the original program P has an error using mutants we will be able to identify the error.

(Refer Slide Time: 28:21)

Mutation testing for source code: Summary



- Mutation testing for source code is one of the most powerful forms of testing source code.
- We need to generate an *effective* set of test cases.
- Next lecture: Mutation operators for typical source code.



What I will do in the next lecture is I will tell you about typical mutation operators that you can use for programming languages like C or Java and that will help you to define mutated programs from ground strings.

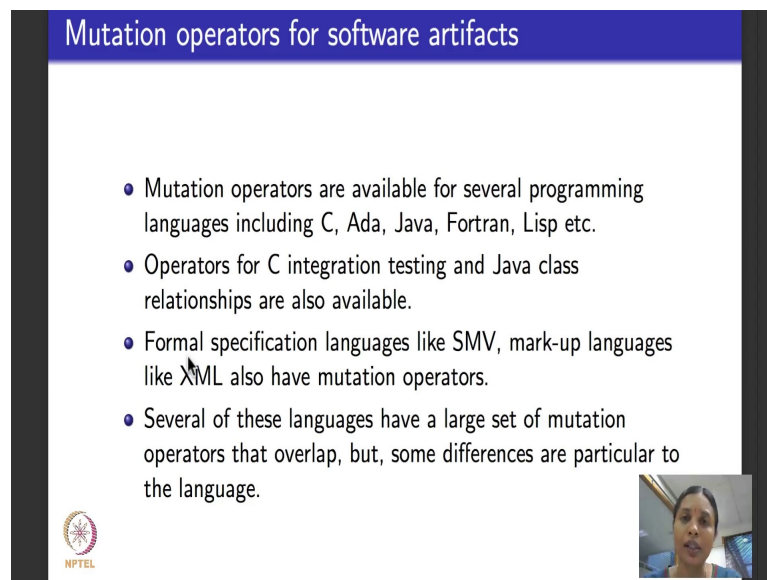
Thank you.

Software testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 38
Mutation testing: Mutation operators for source code



Hello again welcome to the fourth lecture of week 8. If you remember we were been doing mutation testing throughout this week. Last lecture I took an example of a small program, we did two methods and I told you how to apply mutation to particular statements in the program, how to write test cases that would kill those mutants. We distinguish between two notions of killing strong feeling and weak killing. We saw examples of both kind, we also saw examples of an equivalent mutant for the first method that we saw. How are these mutation operators obtained, how these statements are mutated? These statements are mutated by using an underlying set of mutation operators that are available for several different programming languages.

(Refer Slide Time: 01:02)



Mutation operators for software artifacts

- Mutation operators are available for several programming languages including C, Ada, Java, Fortran, Lisp etc.
- Operators for C integration testing and Java class relationships are also available.
- Formal specification languages like SMV, mark-up languages like XML also have mutation operators.
- Several of these languages have a large set of mutation operators that overlap, but, some differences are particular to the language.

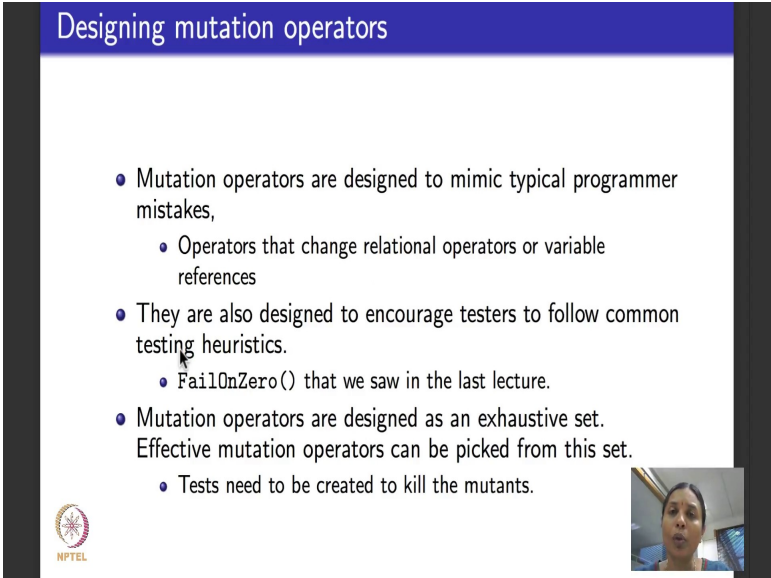
 

So, the focus of this course is to understand mutation operators. As I have been telling you mutation operators are available for many programming languages like C, Java, Ada, Fortran, Lisp and so on. They are available for integration testing of C which focuses purely on procedures calling each other. They are available for integration testing of Java

which focuses on class integration testing, methods calling each other and so on which we will also see.

Then we will come to specification, mutation operators are available for formal specification and modeling languages like SMV and NuSMV. They are available for markup languages like XML which we will see as a part of this course. Some, each of these languages we know is different, it forms slightly different subset, has different syntax and more or less the same set of programs can be written with any programming language. So, what we will see today is like a common set or a generic set of mutation operator that cuts across all the different programming languages and we will also see mutation operators that apply at the individual statement level within a program.

(Refer Slide Time: 02:06)



The slide is titled "Designing mutation operators" in a blue header. It contains a bulleted list of three main points, each with sub-points. The first point is "Mutation operators are designed to mimic typical programmer mistakes," with a sub-point "Operators that change relational operators or variable references." The second point is "They are also designed to encourage testers to follow common testing heuristics," with a sub-point "FailOnZero() that we saw in the last lecture." The third point is "Mutation operators are designed as an exhaustive set. Effective mutation operators can be picked from this set," with a sub-point "Tests need to be created to kill the mutants." In the bottom left corner is the NPTEL logo, and in the bottom right corner is a small video inset showing a person speaking.

- Mutation operators are designed to mimic typical programmer mistakes,
 - Operators that change relational operators or variable references
- They are also designed to encourage testers to follow common testing heuristics.
 - FailOnZero() that we saw in the last lecture.
- Mutation operators are designed as an exhaustive set. Effective mutation operators can be picked from this set.
 - Tests need to be created to kill the mutants.

So, these are not the operators that are going to focus on integration testing. As I told you here they are going to focus on unit testing. We are going to mutate individual statements in the program by making small changes to each statement in a program as and when it is necessary to test the execution of a program. What are these mutation operators designed for? Typically programmers when they unit test their code, why do they do unit test? It unit testing helps them to detect simple mistakes that these programmers make themselves. For example, instead of writing less than or equal to they could have written less than, instead of writing array index as beginning from 0 they

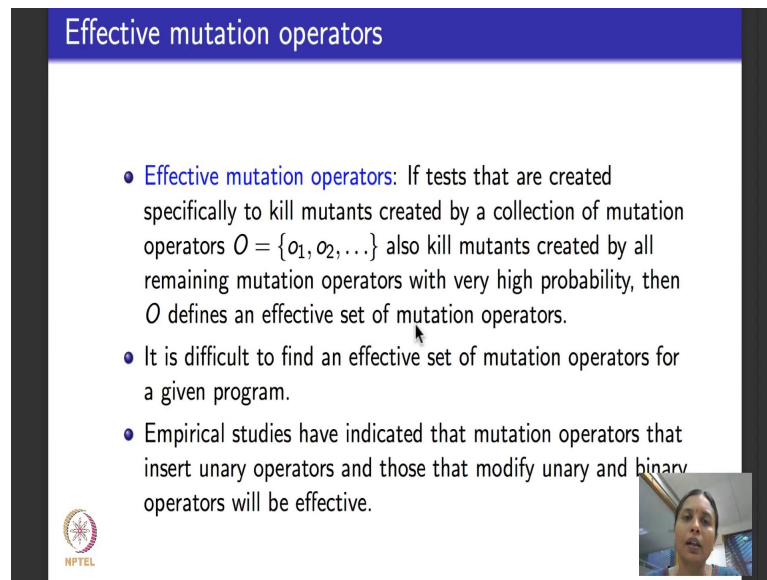
could have written as array index as beginning from one and so on. So, these are simple programmer mistakes.

And what we are going to see is a reasonably exhaustive list of mutation operators that try to mimic or mutate to mimic these simple programmer mistakes. These could be operators that change relational operators as I told you ,maybe he suppose to do less than or equal to, but instead the programmer did less than. Or maybe greater than or equal to instead he did greater than, or maybe he had to do i plus plus instead of that he did j plus plus so that he changed variable name or a variable reference.

So, we are going to see mutation operators that mimic these kind of mistakes they make small changes that mimic these mistakes and see how the program behaves if the mutant can be killed or not. They are also a design to encourage testers to follow common testing heuristics. What do we mean by that? One common testing heuristic which every individual tester is suppose to do see is to see how the individual method that he or she has written, where in the variables in that method become 0, is it fine. Because maybe somewhere in the code there is a division by 0 or there is division by some number that could potentially become 0. So, every programmer the ones on each programmer to test their code to see if all exceptions including variable names being 0 are being handled correctly.

So, for that we will see a specific mutation operator which we already used in the last lecture called fail on 0. Mutation operators that we will see will be fairly exhaustive in the sense that it will be like a laundry list of all possible different mutations that you can do to a particular statement. Typically most of the times it is not the case that a programmer has to use each of the kind of mutation operators that we will be presenting today to be able to test his or her code. But as a list the onus on these tools and on us to be able to provide an exhaustive list. We never know which of these operators from the list is going to be useful to a programmer for his or her code. But this list is exhaustive, but the use is not meant to be exhaustive. Use is meant to be selective as and when needed for specific pieces of code for unit testing.

(Refer Slide Time: 05:26)



The slide is titled "Effective mutation operators" in a blue header. It contains a bulleted list of three points. The first point defines "Effective mutation operators" as a set $O = \{o_1, o_2, \dots\}$ that kills mutants created by all remaining mutation operators with very high probability. The second point states it is difficult to find an effective set of mutation operators for a given program. The third point mentions empirical studies indicating that mutation operators that insert unary operators and those that modify unary and binary operators will be effective. In the bottom left corner is the NPTEL logo, and in the bottom right corner is a small video inset showing a person speaking.

- **Effective mutation operators:** If tests that are created specifically to kill mutants created by a collection of mutation operators $O = \{o_1, o_2, \dots\}$ also kill mutants created by all remaining mutation operators with very high probability, then O defines an effective set of mutation operators.
- It is difficult to find an effective set of mutation operators for a given program.
- Empirical studies have indicated that mutation operators that insert unary operators and those that modify unary and binary operators will be effective.

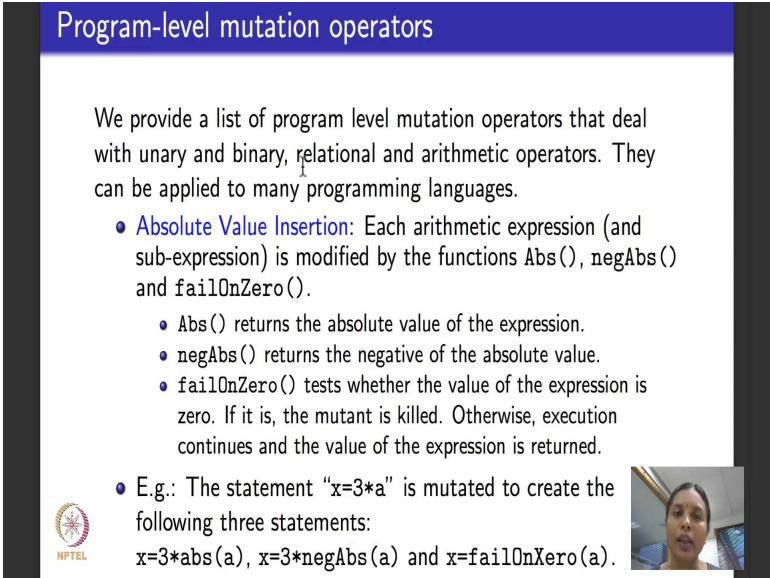
And while mutating a program and testing it the programmer has to create tests to be able to kill the mutant, either strongly kill it or weakly kill it. So, when I said in this exhaustive list of mutation operators, how will a test of pick which are effective mutation operators? How do I know which set of operators are effective. So, here is a definition of effective mutation operators. It says if tests are created specifically, let us say you start with a collection of mutation operator, call that set as the set O , given as O_1, O_2, O_3 and so on, some finite set of mutation operators. You have made these mutations one at a time to the given piece of program and you are writing test cases to kill these mutation operators.

If the test cases that you have killed, written to kill, these mutation operators also happened to kill other unused mutation, the remaining mutation operators with very high probability. You may or may not know whether they were actually killed, but let say they happen to kill the other remaining operators with very high probability, then you say that the underlying set of mutation operators that a programmer has to picked is turning out to be effective. So, just to repeat it, a programmer has a method with him and now the idea is to apply some set of mutation operators to test that individual method written by the program. Let say the programmer picks let say three different set of mutation operators to apply from the exhaustive listing.

So, when do we call these 3 mutation operators is being effective? Let us say the programmer after testing with a reference to these 3 mutation operators also happened to realize that these 3 mutation operators also happen to kill other mutation operators that he did not apply, then you say that the 3 mutation operators chosen by the programmer is an effective set of mutation operators. So, it will be a great thing if per se given a method or a piece of language, if you could have algorithms that determine right up front which are the effective set of mutation operators. Unfortunately it is a difficult problem, an NP complete or sometimes an undecidable problem to be able to determine an effective setup mutation operators, even at the level of an individual method.

But typically various empirical studies in software engineering have indicated that mutation operators of this kind, those that insert unary operators, and those that modify unary and binary operators, have proven to be very effective from the point of view of this definition. So, that is the list that we are going to see in this lecture today.

(Refer Slide Time: 07:59)



Program-level mutation operators

We provide a list of program level mutation operators that deal with unary and binary, relational and arithmetic operators. They can be applied to many programming languages.

- **Absolute Value Insertion:** Each arithmetic expression (and sub-expression) is modified by the functions `Abs()`, `negAbs()` and `failOnZero()`.
 - `Abs()` returns the absolute value of the expression.
 - `negAbs()` returns the negative of the absolute value.
 - `failOnZero()` tests whether the value of the expression is zero. If it is, the mutant is killed. Otherwise, execution continues and the value of the expression is returned.
- E.g.: The statement "`x=3*a`" is mutated to create the following three statements:
`x=3*abs(a)`, `x=3*negAbs(a)` and `x=failOnZero(a)`.

So, what we do now is that I will provide a list of program level mutation operators. What do they deal with? They deal with unary, binary, both relational and arithmetic and logical operations. They cut across several different programming languages. They can be applied for C, for Java or for any other programming language that has a similar syntax. And I will be giving you an exhaustive list of mutation operators, there will be so many different ones. Please remember that you need not apply, given a piece of program

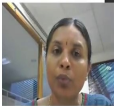

or a method you need not apply every kind of mutation operators. Choose a small subset from this such that you can make use of this list of mutation operators to be able to effectively mutate your program in test.

It is useless to pick everything from this list because it is a fairly large list. So, what are we what are the exhaustive large list of mutation operators that at the program level that we are going to see. So that is what will the rest of the lecture be. So, I will give each of these collection of mutation operators that we are going to see a name like this. That name will be written in blue in every slide. So, it just tells you what is the kind of mutation that we are going to apply. The first collection of mutation operator that we are going to see is what is called absolute value insertion.

(Refer Slide Time: 09:19)

Program-level mutation operators

- **Arithmetic Operator Replacement:** Each occurrence of one of the arithmetic operators $+$, $-$, $*$, $/$, $**$ and $\%$ is replaced by each of the other operators.
- In addition, each is replaced by the special mutation operators *leftOp*, *rightOp* and *mod*.
 - *leftOp* returns the left operand (the right is ignored).
 - *rightOp* returns the right operand.
 - *mod* computes the remainder when the left operand is divided by the right.
- E.g.: The statement $x = a+b$ is mutated to create the following seven statements:
 $x = a-b$, $x = a*b$, $x = a/b$, $x = a**b$, $x = a$, $x = b$
 $x = a \% b$.





We will see what it is. Later I will tell you set of mutation operators called arithmetic operator replacement, we will after that do relational operator replacement, after that do conditional operator replacement, after that do shift operator replacement and so on, we will see a fairly large list.

(Refer Slide Time: 09:25)

Program-level mutation operators

- **Relational Operator Replacement:** Each occurrence of one of the relational operators ($<$, $>$, \leq , \geq , $=$, \neq) is replaced by each of the other operators and by *falseOp* and *trueOp*.
 - *falseOp* always returns false and *trueOp* always returns true.
- E.g.: The statement `if (m>n)` is mutated to create the following seven statements:
`if (m≥n)`, `if (m<n)`, `if (m≤n)`, `if (m==n)`, `if (m=n)`,
`if (false)`, `if (true)`.

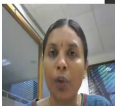



So, each slide will have one collection of mutation operators that belong to one particular category, and remember typically when you test a program only a small number from this exhaustive set is enough to test a program.

(Refer Slide Time: 09:29)

Program-level mutation operators



- **Conditional Operator Replacement:** Each occurrence of each logical operator (and- $\&\&$, or- $\|\|$, and with no conditional evaluation- $\&$, or with no conditional evaluation- $\|$, not equivalent- \wedge) is replaced by each of the other operators. In addition, each is replaced by *falseOp*, *trueOp*, *leftOp* and *rightOp*.
 - *leftOp* returns the left operand (the right is ignored) and *rightOp* returns the right operand (the left is ignored).
- E.g.: The statement `if (a && b)` is mutated to create the following eight statements:
`if (a ||b)`, `if (a &b)`, `if (a |b)`, `if (a ^b)`, `if (false)`, `if (true)`, `if (a)`, `if (b)`.



(Refer Slide Time: 09:32)

Program-level mutation operators

- **Shift Operator Replacement:** Each occurrence of one of the shift operators `<<`, `>>` and `>>>` is replaced by each occurrence of the other operators. In addition, each is replaced by the special mutation operator *leftOp*.
 - *leftOp* returns the left operand unshifted.
- E.g.: The statement `x = m << a` is mutated to create the following three statements:
`x = m >> a`, `x = m >>> a`, `x = m`.



So, we begin with our first set which we call absolute value insertion. So, what we do here? Let us take a statement, if you look at this example here down the slide, let us take a statement like `x is equal to 3 star a`. It is a normal assignment statement. What I am going to do is that I am going to change this to replace the variable `a`, with its absolute value call it as `abs of a`, I am going to replace the variable `a` with a negation of its absolute value this is called `neg abs of a`, and I am going to replace `a` with 0 through the mutation operator `fail on 0`. So, for this statement which looks like this `a` is an absolute value that `a` is a variable, whose which, for which I am mutating by considering 3 possible different mutation operators. Instead of `a` I take the absolute value of `a` that is this first mutated statement that I get. The second one instead of `a`, I take the negation of the absolute value of `a` that is the second mutated statement. The third one instead of `a`, I do `fail on 0` of `a` which makes `a` as 0. So, those are the 3 mutation operators.

So, this I can do for any statement and given any program. So, I say whenever I find an arithmetic expression or a sub expression, and I find a variable in that arithmetic sub expression then I can apply 3 different possible mutations to that variable. One possible mutation returns the absolute value of that expression or the variable, one possible mutation returns the negation of the absolute value of the expression of variable, the third possible mutation tests whether the value of the expression is 0 or not. If it is happen to be 0 then we say that the mutant `fail on 0` is killed, otherwise the mutant is not killed, the

execution continues and the value of the expression as it was originally in the program is returned.

So, the next set of mutation operators that we are going to consider belong to this category of operators called arithmetic operator replacement. If you see in a typical programming language, what are the arithmetic operators that are there? There is addition, plus, subtraction, minus, multiplication, star, division, exponent, mod. So this is an reasonably exhaustive list of possible arithmetic operators. Now if you go down and look at the example, let us say there is a statement that look like this: x is equal to a plus b. It has one arithmetic operator in it, this is the plus. So, I could create 7 different mutations of the statement based on my need, where I replace this arithmetic operator plus with either minus or star or slash, which is division or exponent. Or there are two other special things where I knock off one of the operands, I retain only the left operand, here I retain only the right operand or I could replace this arithmetic operator with a mod. So that is what it says. It says take any statement which has an arithmetic operator in that each occurrence. So, this example the arithmetic operator that we begin with this plus, but it could be the case that the statement original statement was x is equal to a minus b, in which case you choose something else in this list of arithmetic operators to mutate it with, I hope that is clear.

So, what it says is that the statement or the expression could have any of these arithmetic operators. So take that arithmetic operator out replace it with some other arithmetic operator from this list based on you need. Two special kind of other mutations can also be done. One mutation is called leftOp, what is it do? It returns only the left operand, the right operand is ignored that is what we did here for this example. If I had a statement x is equal to a plus b , if I apply the mutation leftOp it results in this, x is equal to a. That is the change that I make the statement to see what would the program do. Similarly rightOp returns just the right operand. So, if I had the left one is ignored. So, if I had x equal to a plus b and I apply rightOp, then I get x is equal to just b, a is ignored and what is mod do mod computes the remainder when the leftOp is divided by the rightOp, the standard thing. So, is that clear.

So, in summary what we saying there could be statements in the program, which have arithmetic operations in them. One possible way to mutate such statements would be replace that arithmetic operator that you find in that expression in that statement. If it is a

plus you could replace it with minus, star, slash, mod and so on. If it is a minus you could replace it with plus, star, slash, mod and so on. So, this collection of mutation operators put together is called arithmetic operator replacement. Moving on, just as arithmetic operators, you could do relational operator replacement. Again what are the exhaustive list of relational operators that are available in programming languages? There is less than, there is greater than, less than or equal to, greater than or equal to, equal to and not equal to.

Let us say there was an expression that look like this. I had an expression which says m is greater than n, I had a predicate sorry which says m is greater than n. My goal, may be I do not know may be the programmer should have written m greater than or equal to n. So, I want to mimic and see is this a mistake paid by the programmer. So, I mutate the predicate m greater than n by replacing the relational operator greater than with greater than or equal to. So, or maybe sometimes the programmer intended to do less than. So, I could mutate the predicate if m greater than n by replacing this relational operator greater than with the relational operator less than. So, what it says is that any occurrence, each possible occurrence of one of the relational operators, which is this list, 6 of them could be replaced by each of the other relational operator. In addition you could do two special mutation operations: one is called falseOp, one is called trueOp. What is falseOp do? FalseOp will replace this whole predicate with false, trueOp will replace this whole predicate with true.

For example if I had an if statement that look like this: if m is greater than n, then I could create seven different mutations from that. I could replace this greater than relational operator with great than or equal to to create this statement if m greater than or equal to n, or I could replace greater than with less than to create the statement, greater than with less than or equal to, greater than with equal to, greater than with not equal to, I am sorry this should be not equal to, then greater than be just false, maybe it is a contradiction, it useful to test it for this, and greater than, m greater than n be just true to test for being a tautology. So, is that clear please. So, what are we saying we say that if I have a predicate, a logical continuing a relational operator, I could create a mutant of that predicate or a clause by replacing the actual relation in the relational operator by any of the other ones that are available. Two special ones, I could replace the whole clause by true, I could replace the whole clause by false.

Moving on, similarly I can do conditional or logical operator replacement. So, what is conditional or logical operator replacement? So, what are the various logical operators? They are and, they are or, and with no conditional evaluation, denoted by a single ampersand, an or with no conditional evaluation which is denoted by a single vertical bar, and not equal to which I denote by this caret symbol. So, I could replace each of these with the other to create so many different mutants. In addition, I also have 4 special mutation operators. What are the special mutation operators? True and false, which are very similar to this one that we saw here, false returns false, true returns true and I have leftOp and rightOp which were similar to the one that we saw in the arithmetic expression rightop and leftOp. So, true returns trueOp returns true, falseOp returns false, leftOp returns the left operand, the right operand is ignored, rightOp returns the right operand and the left operand is ignored.

So, for example, if I had a predicate that looked like this and an if statement with that predicate. Let say if a and b then I could mutate the predicate by replacing this relational operator and with any of the other ones. So, I could replace and with an or to get this, I could replace and with, an and with no conditional evaluation to get the second expression here, I could replace this and with an or with no with conditional evaluation to get the third one. This is replacing and with not equivalent, this is the fourth one, this is replacing and with false, this replacing a and b with true, this is knocking of the b from a this is knocking of the a from a and b. I hope this is clearly right. It is a fairly routine thing, I take one operator of one kind I can substitute it with any other operator of any other kind. Now you might ask why cannot I replace this and with e with let say plus? That would not make sense right. Why, because the type of a and b is meant to be Boolean and I should replace one operator with an another operator of the same category. Please remember that the mutants that I create from programs still have to be valid in the sense that the resulting program has to compile.

So, arithmetic operators that we saw here can be replaced with other arithmetic operators only. Conditional operators, relational operators can be replace to the other relational operators only. Conditional operators can be replace with conditional operators only and so on for all the other groups that we are going to see for the remaining part of this lecture also. So, the next set of mutation that you could have do is regarding the shift operators. You might have seen this left shift, right shift and right shift has two categories

signed and unsigned. So, this is the left shift operator, this is the signed right shift operator, this is the unsigned right shift operator. Unsigned right shift is the counterpart of left shift, signed right shift is meant to take care of negative numbers also.

So, each occurrence of these shift operators can be replaced by the other occurrence. In addition I have this special mutation operator which just returns the left operand which is unshift. So, if I had a statement like this which says x is equal to m left shift a , I can mutate this statement to create 3 different mutations. I can replace this left shift with unsigned, signed right shift, I can replace with this left shift with unsigned right shift or I can replace this left shift by knocking off the left shift and the a , do not shift and return only m . So, these are 3 mutations that are possible when you have statements in your program that deal with shift operations.



Moving on, suppose you had statements in your program that deal with bitwise logical operators. That is, they work a lot like and, or, but when given a sequence of bits or binary numbers, they and, or, or exclusive or, bit by bit by applying the same truth table for and or. So, each occurrence of each of the bitwise operators the bitwise and the bitwise or the bitwise exclusive or, can be replaced by each of the other operators when we do mutation operators here. In addition I can also do leftOp and rightOp like I did for arithmetic and other categories. LeftOp always returns the left operand right one is ignored, rightOp returns the right operand left one is ignored.

So, suppose I had a statement like this: x is equal to m bit wise and n , then I can create 3 possible mutants of the statements. I could replace this bitwise and by bitwise or, this is a first mutant, I could replace this bitwise and by bitwise exclusive or, the second mutation, I could knock off the second left, the right operand and consider only x is equal to m , I could knock off the left operand and consider only x is equal to n .

(Refer Slide Time: 22:16)

Program-level mutation operators

- **Assignment Operator Replacement:** Each occurrence of one of the assignment operators (`+=`, `-=`, `*=`, `=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`, `>>>=`) is replaced by each of the other operators.
- E.g.: The statement `x += 3` is mutated to create the following ten statements:
`x -=3, x *=3, x =3, x /=3, x &=3, x |=3, x ^=3, x <<=3, x >>=3, x >>>=3.`



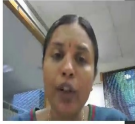

So, these kind of mutations are possible dealing with bitwise operations. Similarly I could also consider these assignment operators plus equal to, minus equal to, star equal to, slash equal to, percentile equal to and so on and I could replace each occurrence of each kind of assignment operator with the other occurrence.

For example if I had a statement which said `x plus is equal to 3`, I can mutate it to create ten different statements. I could say `x minus equal to 3`, `x star equal to 3`, `x slash equal to 3`, `x slash equal to 3`, `x a mod equal to 3` and so on and so forth.

(Refer Slide Time: 22:49)

Program-level mutation operators

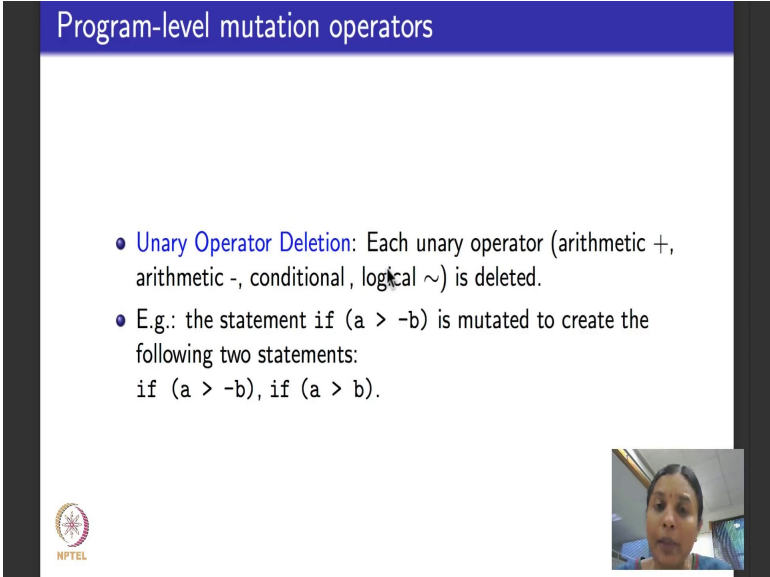
- **Unary Operator Insertion:** Each unary operator (arithmetic `+`, arithmetic `-`, conditional, logical `~`) is inserted before each expression of the correct type.
- E.g.: the statement `x = 3 * a` is mutated to create the following four statements:
`x = 3 * +a, x = 3 * -a, x = +3 * a, x = -3 * a.`



Now the other kind of mutation that I can do is unary operator insertion. What do we mean by that? Let us begin by understanding the example. Suppose there is a statement which says x is equal to 3 into a. Now I can replace this a with minus with plus a, create one mutation, this mutation turns out to be equivalent to this because if a is positive then plus a has the same sign as a, if a is negative also then the same thing. So, I list this for the sake of completeness, but these two are equivalent. Then I could replace this a with minus a create another mutation.

Similarly, I could replace 3 with plus 3. Here again it is just an equivalent mutant might as well not be created because it gives the same expression, but it is listed here for the sake of being exhaustive. I could replace this 3 with a minus 3 to create such an expression. So, that is class of mutation operators is what is called unary operator insertion. What do I do here? Each unary operator which is the arithmetic plus, the arithmetic minus, the conditional and the logical is inserted before each expression of the correct type. Here I had arithmetic expression. So, I am considering only the arithmetic unary operators. Later I could consider the logical or the conditional unary operators.

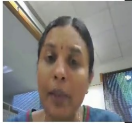
(Refer Slide Time: 24:07)



Program-level mutation operators

- **Unary Operator Deletion:** Each unary operator (arithmetic +, arithmetic -, conditional, logical \sim) is deleted.
- E.g.: the statement `if (a > -b)` is mutated to create the following two statements:
`if (a > b), if (a > b).`

NPTEL




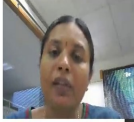
Just as replacing, I could delete a unary operator all together. Each unary operator can be deleted, it could be the plus, it could be the minus, it could be a conditional operator, it could be a logical negation I can delete it.

For example, here I say if a is greater than minus b, I could say a is greater than minus b which is the same as this or I could remove the minus b and say a is greater than b. So, I could remove that. Now this is a very useful thing which we also used in the example that we saw in the last section. There are variables where I find an expressions throughout my piece of program.

(Refer Slide Time: 24:33)

Program-level mutation operators

- **Scalar variable replacement:** Each variable reference is replaced by every other variable of the appropriate type that is declared in the current scope.
- E.g.: the statement $x = a * b$ is mutated to create the following six statements:
 $x = a * a$, $a = a * b$, $x = x * b$, $x = a * x$, $x = b * b$, $x = a * b$.





And those variables are called scalar variables because they are not as vectors, there is one individual variable. What I can do is I can mutate the program by considering each variable reference and replace it with some other variable reference such that the type matches. So, the expression continues to get evaluated and not only the type should match, the other variable reference that I replace this with should be within the scope, that is what is true this thing. I cannot pick up a another variable far away somewhere else in the program and suddenly substitute it here. It should be within the scope because if it is within this scope then things like declaring the variable and all other things will be taken care of, and mutating this way is effective than replacing with a random undeclared variable.

So for example if I had a statement x is equal to a star b, then what can I do? I can take this and replace this b with a to create an expression like this, x is equal to a star a. I could replace the x with a to create another expression like this, I could replace the a with x to create this third expression. I could replace b with x to create the forth



expression, I could replace a with b to create this expression and finally, I retain the same thing, is that clear. There is no point now in picking up a completely new variable let say c that is out of the scope of this statement and replace x is equal to c in to b maybe the program will complain saying c is not been declared and so on. So, you do not want to risk. Remember programs have to compile, they have to be valid. So, when I do always replacement I am always replace variables such the type of the variables match at sorry and the variable that is declared is within the current scope.

(Refer Slide Time: 26:29)



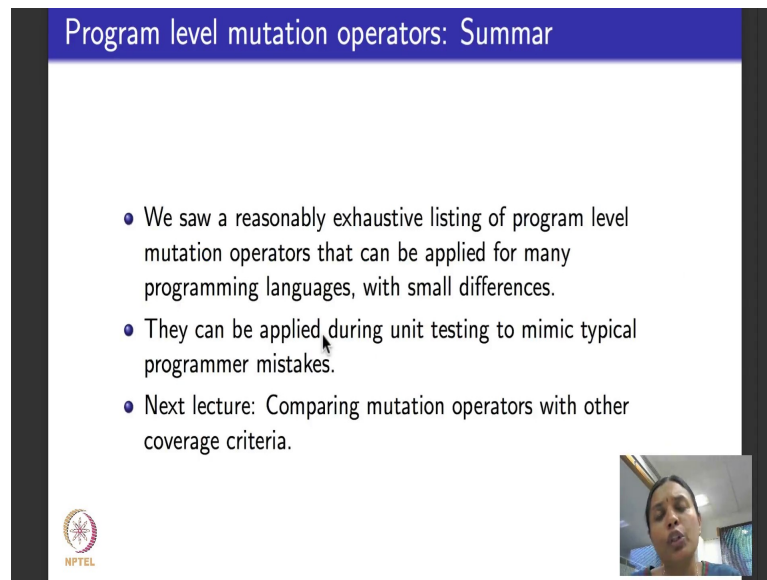
Program-level mutation operators

- **Bomb Statement Replacement:** Each statement is replaced by a special Bomb() function.
- Bomb() signals a failure as soon as it is executed, thus requiring the tester to reach each statement.

Finally we saw this last time. There is a special mutation available called bomb statement what is this bomb statement do? Bomb signals a failure as soon as it is reached. It is like making the program fail at that statement and see what happens. This is another common testing heuristics which many companies that recommend unit tester has to do. So, they say reach each statement. So, one way to make sure that the program execution has reached a particular statement is to replace that statement with a bomb. When the program reaches that statement, it will terminate. So, you know that the program is reached that statement.

(Refer Slide Time: 27:07)



Program level mutation operators: Summar

- We saw a reasonably exhaustive listing of program level mutation operators that can be applied for many programming languages, with small differences.
- They can be applied during unit testing to mimic typical programmer mistakes.
- Next lecture: Comparing mutation operators with other coverage criteria.

NPTEL

A small video inset in the bottom right corner shows a person speaking.

So, bomb is a very useful mutation operator. So, what did we do in this lecture? We saw an exhaustive list. Please remember the word is exhaustive, there is no point in considering each and every mutation operator that we saw today for every program that you test for. You should carefully select from this list, which is the one that I want to test in my program, which is the property, which is the statement.

If I want to test this statement for one particular property which would be a good mutation operated to pick from the list that we learnt today, that is how you should use it. And typically it is applied during unit testing to mimic typical programmer mistake as I told you. In the next lecture what I will be telling you is we saw mutation testing and one question that might naturally arise in your mind is, how does mutation testing compare to graph testing or how does it compare to logical predicate testing because they were also white box testing techniques right. So, that is what we will see, we see that mutation testing in fact is very powerful, it subsumes a lot of criteria from graph base testing and from logical base testing.

Thank you.

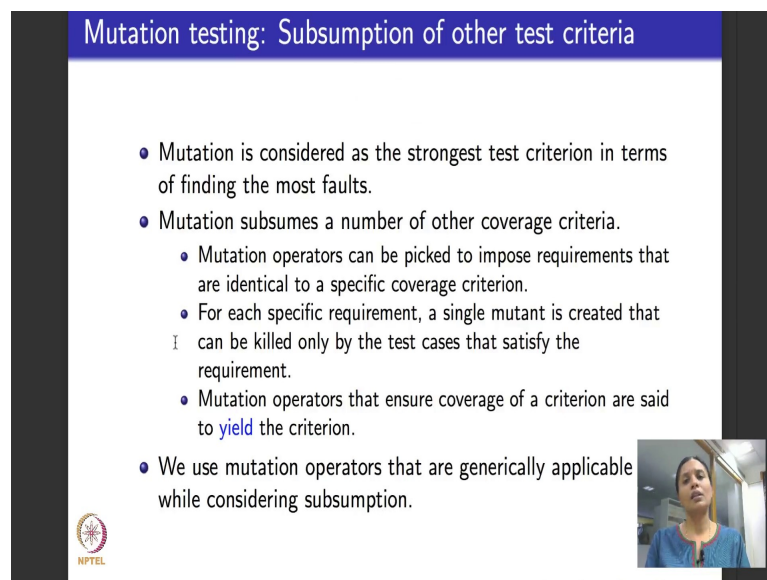
Software testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 39
Mutation testing vs. graphs and logic based testing

Hello there, we are in the last lecture of week 8. We have been doing mutation testing. I told you, began this week, by introducing you to regular expressions and grammars. Then we introduce generic terms and mutation testing. Then we applied mutation testing to source code we saw 2 examples of that. In the last lecture I gave you a reasonably exhaustive list of mutation operators that you could use to mutate a program for doing unit testing, within a method, within a piece of code, that lets you make changes to the statements in a program.


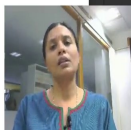
Next week we will apply mutation testing for design integration. But before we do that, as I ended my last lecture, as I told you, we saw mutation testing, before this we saw graph testing logical predicate testing. How does mutation testing compare to all the other testing criteria that we saw?

(Refer Slide Time: 01:08)



Mutation testing: Subsumption of other test criteria

- Mutation is considered as the strongest test criterion in terms of finding the most faults.
- Mutation subsumes a number of other coverage criteria.
 - Mutation operators can be picked to impose requirements that are identical to a specific coverage criterion.
 - For each specific requirement, a single mutant is created that can be killed only by the test cases that satisfy the requirement.
 - Mutation operators that ensure coverage of a criterion are said to **yield** the criterion.
- We use mutation operators that are generically applicable while considering subsumption.

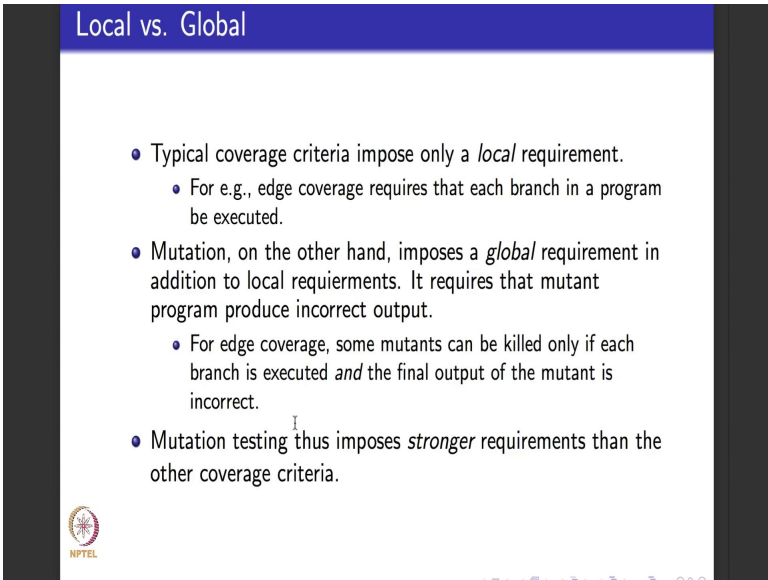
 

I told you that mutation testing is considered as one of the strongest test criteria in terms of finding, capability of finding the most number of faults. So, if you go by that then it should subsume a good amount of graph based testing, and a good amount of logical

predicate based testing also. So, in this lecture we will see which are the graph criteria that mutation testing is going to subsume, which are the logical predicate criteria that mutation testing is going to subsume.

So, mutation operators can be picked to be subsume the kind of criteria we want. Let us say suppose we want to make a claim that mutation is testing subsumes edge coverage. Then I say, in the control flow graph to achieve edge coverage for the program, what are the mutation operators that I should pick in the program for this to happen? So, I pick my mutation operators to make mutation testing subsume the kind of coverage criteria I want to do. For certain coverage criteria it is possible to pick mutation operators to do this. For certain criteria it is not possible to pick mutation operators that do this. So, for criteria for which it is not possible to pick mutation operators to do this, mutation testing does not subsume those kind of criteria. Mutation operators that ensure that a particular criteria is covered put together are called, what are called, that make the mutation testing subsume the coverage criteria, what are called yielding the criteria. So, we use mutation operators that are generically applicable, not specific to any programming language, not specific to any entity that way when we consider subsumption.

(Refer Slide Time: 02:50)



The slide is titled "Local vs. Global" in a blue header. It contains a bulleted list comparing local and global requirements for coverage criteria. The first bullet point states that typical coverage criteria impose only a *local* requirement, with an example that edge coverage requires each branch to be executed. The second bullet point states that mutation imposes a *global* requirement in addition to local requirements, requiring the mutant program to produce incorrect output. A sub-bullet for edge coverage notes that some mutants can only be killed if each branch is executed *and* the final output is incorrect. The third bullet point concludes that mutation testing imposes *stronger* requirements than other coverage criteria. An NPTEL logo is in the bottom left corner, and navigation icons are in the bottom right.

Local vs. Global

- Typical coverage criteria impose only a *local* requirement.
 - For e.g., edge coverage requires that each branch in a program be executed.
- Mutation, on the other hand, imposes a *global* requirement in addition to local requirements. It requires that mutant program produce incorrect output.
 - For edge coverage, some mutants can be killed only if each branch is executed *and* the final output of the mutant is incorrect.
- Mutation testing thus imposes *stronger* requirements than the other coverage criteria.

NPTEL

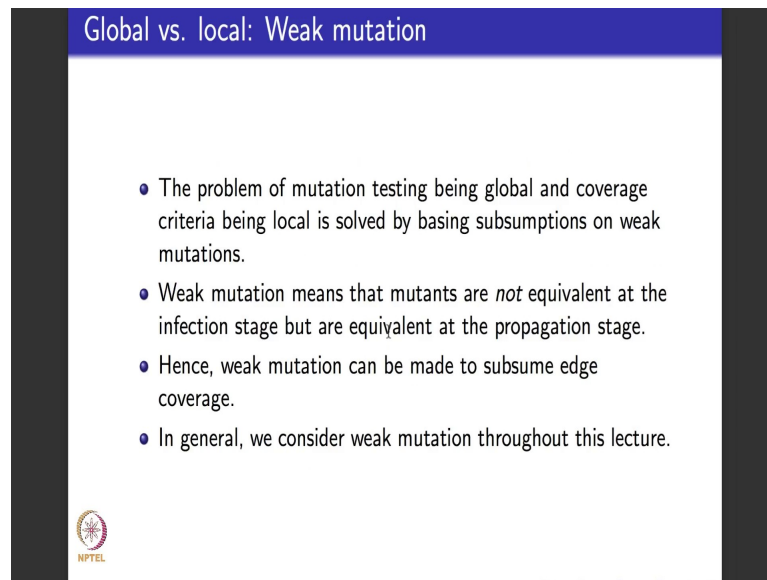
So, if you look at coverage criteria like graph coverage criteria or logical coverage criteria, you will realize that they are somewhat local coverage criteria. They impose only a local requirement. What do we mean by a local requirement? Let us take an

example. You remember we saw edge coverage when we did graph based coverage, and we later realized that predicate coverage and logic coverage is the same as edge based coverage. What is that require? What is the local requirement that it imposes? It says that every branch in a program be executed. It does not talk about anything related to inputs and outputs of a program. That is what mutation testing talks about. But edge coverage says in the program, in the control flow graph of the program execute every edge.

So, it talks about a requirement that is local to the edges of the control flow graph of a program. But mutation testing imposes a global requirement in addition to the local requirement. It requires that the mutated program produce an incorrect output. We saw notions of weak mutation and strong mutation, if you remember. Weak mutation does not need a program to produce incorrect input, it requires that infection happens at that statement. But strong mutation needs a program, I mean strong killing a mutation needs a program to produce a different output. So, that is what is called a global requirement.

So, in some sense because of this, mutation testing imposes stronger requirements on the software coverage than other coverage criteria. The problem of mutation testing being global and coverage criteria being local, how do you solve it? As I told you when I say mutation testing is global back in this slide, why do we I mean by global? We mean that it is global because we want the mutant to produce an incorrect output, which means in the reachability, infection, propagation model, I want the mutant to ensure conditions of reachability of infection, and of propagating all the way to the outputs to produce a different output. We later refine this notion of killing to say that this is actually strongly killing the mutant. We could have the option of weakly killing mutant by which we satisfy only reachability and infection need not propagate.

(Refer Slide Time: 05:00)



Global vs. local: Weak mutation

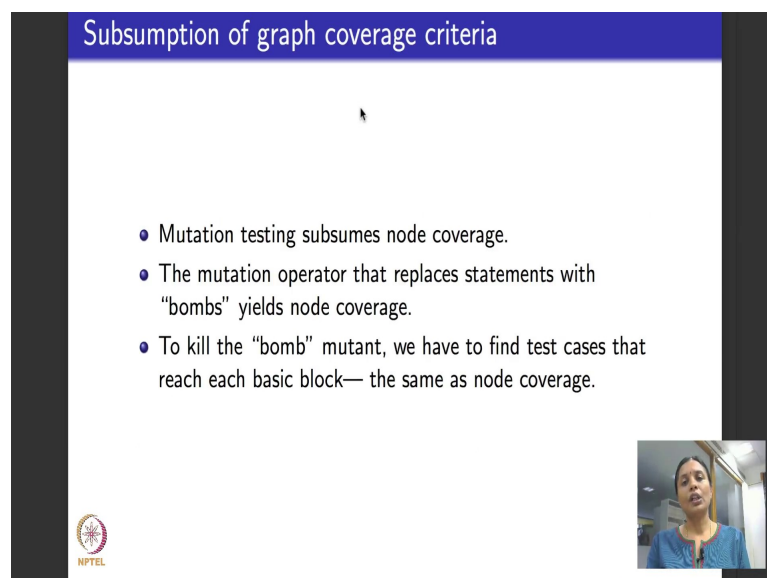
- The problem of mutation testing being global and coverage criteria being local is solved by basing subsumptions on weak mutations.
- Weak mutation means that mutants are *not* equivalent at the infection stage but are equivalent at the propagation stage.
- Hence, weak mutation can be made to subsume edge coverage.
- In general, we consider weak mutation throughout this lecture.

NPTEL

So, that is the notion of mutation that we will use through most of this lecture except towards the end when we look at data flow criteria.

So, we consider weak mutation means mutants are not equivalent at the infection stage, but can be equivalent at the propagation state. So, weak mutation can be thought of us making mutation testing local to the statement or the set of statements that we are considering in the graph. So, using weak mutants we can compare mutation testing to other coverage criteria.


(Refer Slide Time: 05:42)



Subsumption of graph coverage criteria

- Mutation testing subsumes node coverage.
- The mutation operator that replaces statements with "bombs" yields node coverage.
- To kill the "bomb" mutant, we have to find test cases that reach each basic block— the same as node coverage.

NPTEL

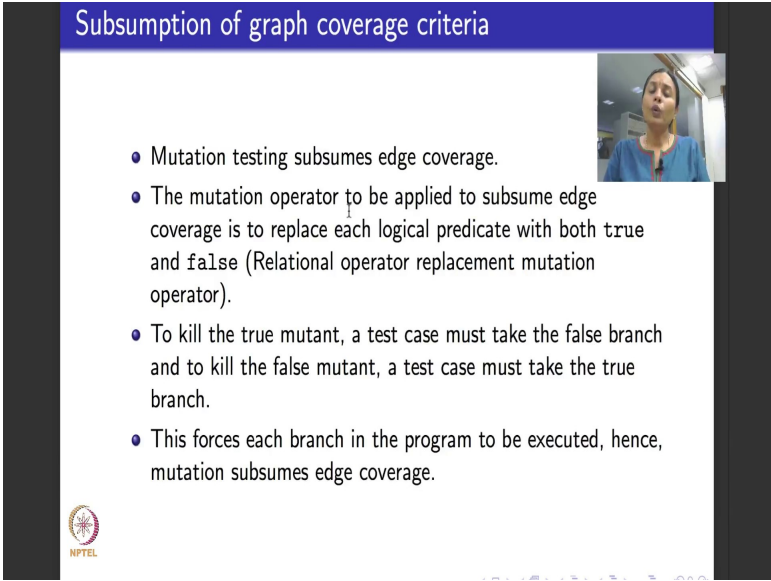


What are the graph coverage criteria that mutation testing is going to subsume? Mutation testing subsumes node coverage. Why does it subsume node coverage? So, node coverage for graphs when it is applied to programs mean, in each or execute every block of statements and every statement in the program. If you remember last lecture I had told you about this bomb mutation operator. Bomb, whenever it is placed at a particular statement, as a particular statement in the program, causes the state program to fail at that point.

So, let say you want to achieve node coverage. What you do, is take a program draw it is control flow graph of a program, you want a test case that will achieve node coverage for this program. How do you do it by using bomb statement? Let say you want to target a node 7 which corresponds to some particular statement in the program. You put a bomb statement in the mutated version at that place in the program and write a test case to kill that mutant. When you write a test case to kill that mutant, the place or the statement or the node at which you put the bomb statement will be visited by the test case, and hence that will be covered by the test case. So, by using bomb statements at a appropriate places in the program, we can ensure the node coverage is met.

So, mutation testing subsumes node coverage criteria.

(Refer Slide Time: 07:02)



The slide is titled "Subsumption of graph coverage criteria" in a blue header. It contains a bulleted list of four points explaining how mutation testing subsumes edge coverage. A small video inset in the top right corner shows a woman speaking. The NPTEL logo is in the bottom left corner.

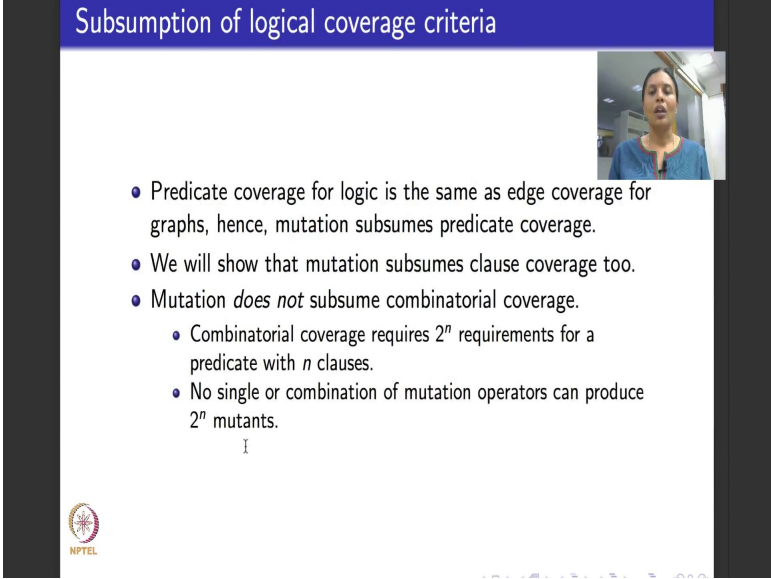
- Mutation testing subsumes edge coverage.
- The mutation operator to be applied to subsume edge coverage is to replace each logical predicate with both true and false (Relational operator replacement mutation operator).
- To kill the true mutant, a test case must take the false branch and to kill the false mutant, a test case must take the true branch.
- This forces each branch in the program to be executed, hence, mutation subsumes edge coverage.

Mutation testing also happens to subsume edge coverage criteria, why is that so? How does edge coverage happen, why do you get different edges in a node? We saw that edge

coverage is the same as predicate coverage. From a particular node in the graph corresponding to a program, we take 2 different edges because that node had a decision statement in the program. One edge results because the decision statement evaluated to true, one edge results because the decision statement evaluated to false. So, I can, it is natural that I use those mutation operators, right? So, I say I will do this is relational operator replacement mutation operator that we saw in the last lecture. I will replace each logical predicate that I encounter in the program with true once to create another mutant. Now if I write test cases to weakly kill these 2 mutants, one test case to take the true edge because I have replace the predicate with true one test case would take the false edge because I have replace the predicate with false. Between these 2 test cases I would have achieved edge coverage for this program.

So, mutation testing does subsume edge coverage.

(Refer Slide Time: 08:15)



Subsumption of logical coverage criteria

- Predicate coverage for logic is the same as edge coverage for graphs, hence, mutation subsumes predicate coverage.
- We will show that mutation subsumes clause coverage too.
- Mutation *does not* subsume combinatorial coverage.
 - Combinatorial coverage requires 2^n requirements for a predicate with n clauses.
 - No single or combination of mutation operators can produce 2^n mutants.

I

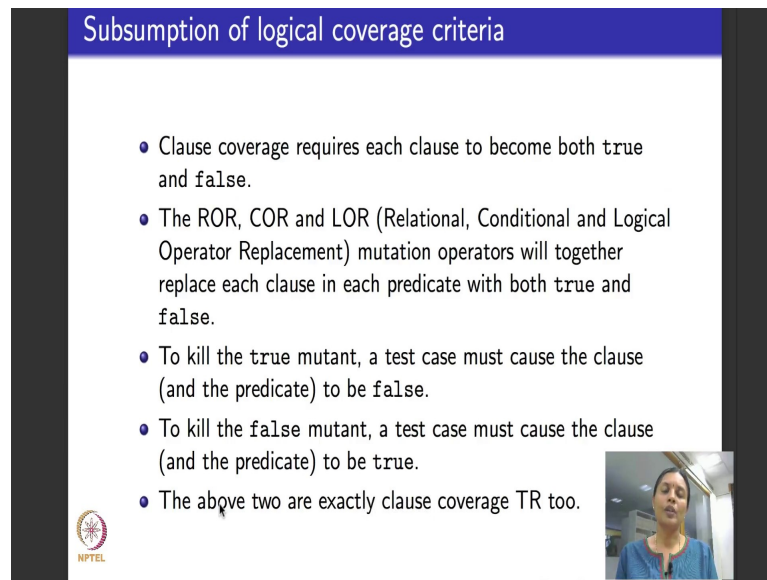
Now let us move on to logical coverage criteria. What are the various logical coverage criteria that mutation testing subsumes? By the way, before we move on, for graph coverage criteria mutation testing just subsumes node and edge coverage. It does not subsume other coverage criteria, like prime path coverage, complete path coverage and all because it does not deal with behavioral aspects of, such behavioral aspects of programs. Now will logic coverage criteria it so happens that mutation testing subsumes clause coverage, predicate coverage, and some amount of active clause coverage also. It

is does not subsume inactive clause coverage and combinatorial coverage. So we will see why.

Predicate coverage for logic, as I told you, is the same as edge coverage for graphs. And just in the previous slide I explained to you about how mutation testing subsumes edge coverage. Since the predicate coverage is the same as edge coverage, mutation testing subsuming edge coverage implies that mutation testing also subsumes predicate coverage. You just have to make the predicate true once, and make the predicate false once. We will show that mutation testing subsumes clause coverage. Before we move on to show that, I would also like to tell you that mutation testing does not subsume combinatorial coverage. You remember what combinatorial coverage is? It says write every possible values of true false for each of the clauses in a predicate and write test cases that will test the predicate for each possible value of true, false values for each of the clauses. So combinatorial coverage because of this each possible combination being considered, we understand requires 2^n requirements for a predicate with n clauses. Because each clause can take to 2 true or false totally n clauses for 2^n possible values can be there. No single or combinational mutation operators can produce 2^n different mutants. It is very exhaustive and difficult, because it is very exhaustive and difficult we say that mutation testing does not subsume combinatorial coverage.



Now, what we will show is why does mutation testing subsume clause coverage.

(Refer Slide Time: 10:29)



Subsumption of logical coverage criteria

- Clause coverage requires each clause to become both true and false.
- The ROR, COR and LOR (Relational, Conditional and Logical Operator Replacement) mutation operators will together replace each clause in each predicate with both true and false.
- To kill the true mutant, a test case must cause the clause (and the predicate) to be false.
- To kill the false mutant, a test case must cause the clause (and the predicate) to be true.
- The above two are exactly clause coverage TR too.

What does clause coverage tell you? If you recall clause coverage says each clause has to be tested to be true once and tested to be false once. If you remember in the last lecture, we saw these 3 categories of mutation testing operators. Relational operator replacement which replaced the less than, less than or equal to, greater than or equal to, and not equal to with each other. Conditional operator replacement which replace the AND or NOTs XORs with each other. Logical operator replacement which replace the bit wise and, bitwise or and bitwise XOR with each other.

Between these 3 mutation operators you could alter each clause in the predicate to be come true once and become false once. Why, because each clause in a predicate will be a combination of these operators, and the blind mutation operator that you could apply is to replace one with true and one with false wherever it is applicable. Wherever it is not then you reverse, you negate the relational operator to make it false, and retain the relational operator as it is to make it true. Suppose there was a clause which says x is less than or equal to y which was a part of a predicate, then x is less than or equal to y can be made true as it is. To make it false you replace the less than or equal to in the clause by using a relational operator replacement of greater than. So, instead of writing x less than or equal to y, you write x greater than y.

So, this is a variant that will make the clause false. So, like this each clause in a predicate can be make true by retaining as it is if it is turns out to be true, and can be made false by

flipping the relational arithmetic or conditional or logical operators that are involved in the clause. So, to kill the true mutant a test case must cause the clause and the predicates to be false, which is what it will do. To kill the false mutant the test case must clause the clause and the predicate to be true because it has to do different output than the original program. And this is exactly what clause coverage does right, because each clause is may true once and false once in this process.

So, mutation testing does subsume clause coverage. I will explain this part the detailed example to you.

(Refer Slide Time: 12:49)

Mutation subsuming clause coverage: Example

Consider the predicate $p = a \wedge b$ along with truth table data for p as below.

	$a \wedge b$	(TT)	(TF)	(FT)	(FF)
1	<i>true</i> \wedge <i>b</i>	T	F	T	F
2	<i>false</i> \wedge <i>b</i>	F	F	F	F
3	<i>a</i> \wedge <i>true</i>	T	T	F	F
4	<i>a</i> \wedge <i>false</i>	F	F	F	F

- To kill mutants, the tester must choose an input (top row of the table) that causes a result that is different from the original predicate.
 - This choice is indicated in bold above.
- Mutant 1 can be killed by the assignment (F T), mutant 3 can be killed by the assignment (T F). Together, they satisfy clause coverage.
- Mutants 2 and 4 are not needed for clause coverage.

Let us consider this predicate p is equal to a and b , it has 2 clauses a and b . So, I have depicted this table in a slightly different way. So let me explain this table to you. So, what it says is this is the predicate here a and b , and right on top here this TT, TF, FT, FF within brackets tell you, that the a clauses a and b in the predicate, both are true in the case of TT, TF means a is true, b is false, FT means a is false, b is true. FF is both a and b are false. Now what did I tell you, I told you here that each clause can be made true once and false once using any of these operators. So, there are 2 clauses here. So, this part, the second column in this table, I had made a true once false once retained b as it is. In this part rows 3 and 4, I have made b true once, false once, retain a as it is.

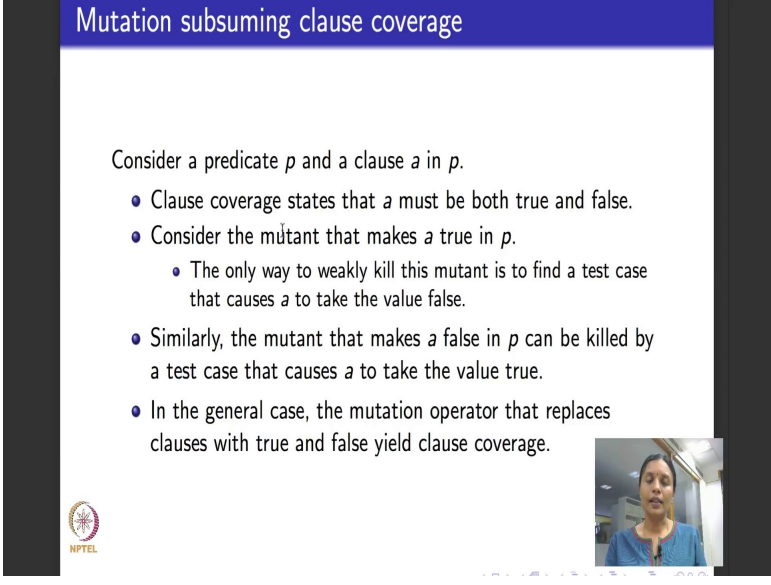
Now, let me see what happens to this. So, what do these mean these mean what is the value that the predicate takes right. So, if both a and b are true, what is true and b take, it

becomes true. If a is true and b is false then, what is true and b become it becomes false. If a is false and b is true there is no a here, but basically dependent on the value of b because b is true, it is true, and now both are false in particular b is false. So, it becomes false. For this part of the table, I have populated as the true false values that these predicates where one clause is replaced with true and false will take. The top part indicates true, false assignments for a and b. This row here indicates what happens to this predicate a and b. Remember to kill the mutants, tester must choose an input which is here, right on top row of this table, that causes the result to be different from the original predicate.

So, let us say this is the first mutant. What have I done in this mutant? I have taken a made it true. So, now, let us see which is the place where this resulting predicate, mutated predicate is different from the original predicate a and b. Here both are true, not different. Here both are false, they are the same again not different. Here the original predicate is false, but the mutated predicate is true. So, the test case that will kill the first mutation makes a false and b true, is that clear? So, this that is why it is been colored in bold here, it is been indicated in bold. Similarly, if I take this as the original predicate and this is the mutated predicate where I replace a with false to be able to kill this mutated predicate, the only test case that will kill will be this one a is true, b is true because the value of the original predicate here is true whereas, the value of the mutated predicate is false.

Similarly, for the other rows. Now if you take mutant one, mutant one can be killed by the assignment false and true, this part indicated by bold as I told you. And mutant 3 can be killed by the predicate true by the assignment a is equal to true, b is equal to false which is indicated as this bold true here, because it differs from the value of the predicate. Between these 2 test cases, if you notice a is made true once, false once. And a is b is made false once, true once. So, it does achieve clause coverage. In fact, we just mutants one and 3, we can achieve clause coverage for this predicate. So, mutation testing does subsume clause coverage. In this example, the mutants 2 and 4 even though I have given them in this table for completeness are not necessary. We might as well not done them at all.



(Refer Slide Time: 16:39)



Mutation subsuming clause coverage

Consider a predicate p and a clause a in p .

- Clause coverage states that a must be both true and false.
- Consider the mutant that makes a true in p .
 - The only way to weakly kill this mutant is to find a test case that causes a to take the value false.
- Similarly, the mutant that makes a false in p can be killed by a test case that causes a to take the value true.
- In the general case, the mutation operator that replaces clauses with true and false yield clause coverage.

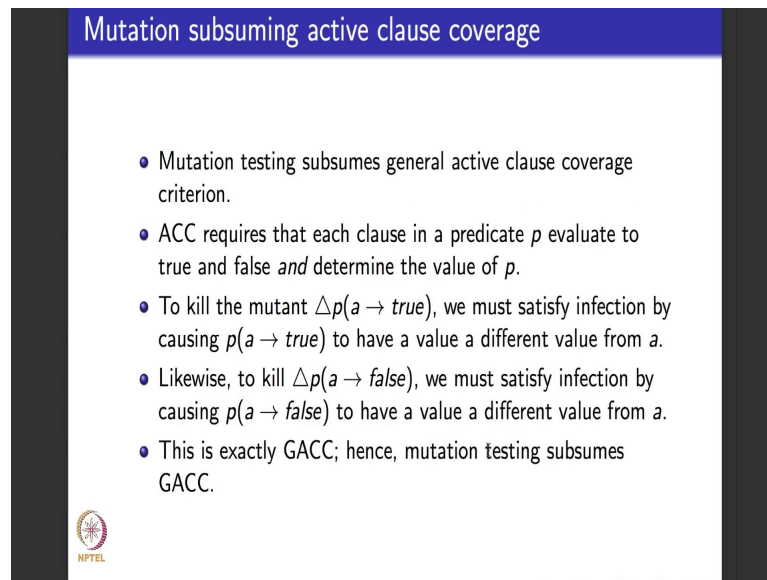
 

Now to generalize this consider a predicate and let a be a clause in the predicate. Clause coverage for a says, a must be true and must be false. Now consider a mutant that makes a true in the predicate. The only way to weakly kill this mutant is to find a test case that causes a to take the value false, is that clear?

Similarly, the mutant that makes a false in the predicate can be killed by a test case that causes a to take the value true. That is what happened in this example right. Here a was made true, the only test case that killed this mutant of this predicate, if you see a was made false. In the third mutant that we considered, b was made true, the only test case that killed the third mutant b here which is this value, is made false as what is written. So, it has to differ in the opposite way. By differing the opposite way it makes sure that each clause that I am considering currently in the predicate is made true once and false once. So, it does manage to achieve cross clause coverage.

So, how do I generalize this? Take any predicate that involves a few clauses. How do I make mutation coverage subsume clause coverage? Just choose a mutation operator that replaces each clause with true and false and then you will get clause coverage immediately.

(Refer Slide Time: 17:55)



Mutation subsuming active clause coverage

- Mutation testing subsumes general active clause coverage criterion.
- ACC requires that each clause in a predicate p evaluate to true and false *and* determine the value of p .
- To kill the mutant $\Delta p(a \rightarrow \text{true})$, we must satisfy infection by causing $p(a \rightarrow \text{true})$ to have a value a different value from a .
- Likewise, to kill $\Delta p(a \rightarrow \text{false})$, we must satisfy infection by causing $p(a \rightarrow \text{false})$ to have a value a different value from a .
- This is exactly GACC; hence, mutation testing subsumes GACC.

NPTEL

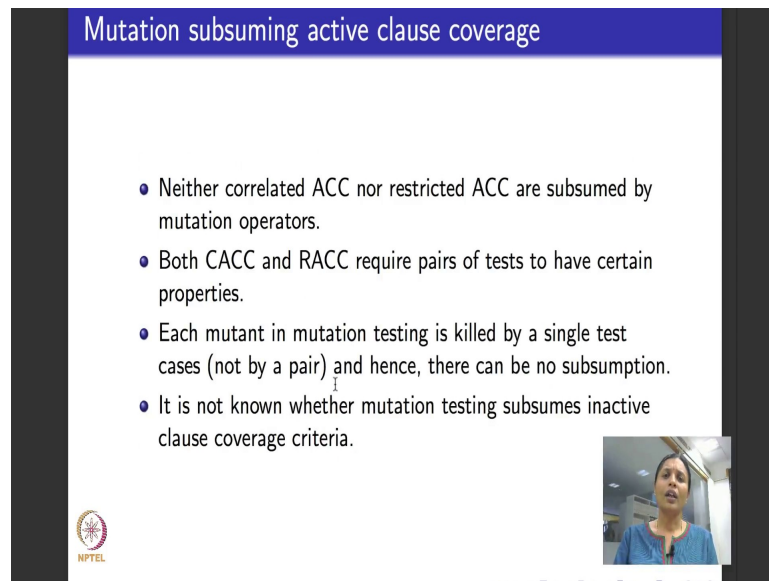
Mutation testing also happens to subsume active clause coverage. If you remember we had seen 3 different active clause coverage criteria when we did logical predicates based testing. The 3 criteria that we saw were generalized active clause coverage criteria, called GACC, correlated active clause coverage criteria called CACC, restricted active clause coverage criteria called RACC. It so happens that mutation testing subsumes only GACC, cannot be made to subsume CACC and RACC. I will tell you why so. So, ACC, what is ACC criteria require? Active clause coverage criteria requires that each clause in a predicate evaluate to true once and false once and while doing so, also determine the value of the predicate. So, which means what? The value that the predicate takes when the clause is true should be different from the value that the predicate takes when the clause is false.

So, to suppose let us say I take the predicate p , please read this phrase that I have written here, as in the predicate p , I mutate or change p by replacing a with the value true, is that clear? So, in the predicate p , I take a , every occurrence of a and p and replace it with true. So, I said this is the mutated predicate with every occurrence of a replaced by true. How will we satisfy infection? We will satisfy infection by causing, this was the original predicate, this is the mutated predicate, these 2 should have different values. Similarly when I take p and replace a with false, we can again satisfy infection by having the original predicate have a different value from the mutated predicate. This is exactly what is generalized active clause coverage criteria. Because it says that each clause becomes

true and false here for example, if a is an arbitrary clause that we have chosen, a becomes true and false once and to kill it the predicate should have a different value in both the cases. And this is exactly the definition of generalized active clause coverage criteria.

Hence mutation testing subsumes GACC.

(Refer Slide Time: 20:08)



The slide has a blue header with the text "Mutation subsuming active clause coverage". Below the header, there is a white box containing a bulleted list of four points. In the bottom right corner of the slide, there is a small video inset showing a person speaking. The NPTEL logo is in the bottom left corner.


- Neither correlated ACC nor restricted ACC are subsumed by mutation operators.
- Both CACC and RACC require pairs of tests to have certain properties.
- Each mutant in mutation testing is killed by a single test cases (not by a pair) and hence, there can be no subsumption.
- It is not known whether mutation testing subsumes inactive clause coverage criteria.

Now moving on, as I told you CACC and RACC are not subsumed by mutation testing or mutation operators cannot be designed to subsume CACC and RACC. Why is that so? If you remember the definition of CACC and RACC, it says while the major clauses determining the truth value of the predicate the minor clauses in one case should all have different values. In the other case, the minor clauses should have all the same values. So, it not only imposes a condition on one clause in the predicate, it imposes conditions on all the other remaining clauses in the predicate. So, they come as pairs of conditions. Each mutant in mutation testing is meant to kill only one condition that can be implemented at a test case not as a pair of conditions. Mutation testing is not meant for that. As I told you the wisest thing to do in mutation testing is to apply one mutation operator, at a time and because we apply one mutation operator at a time we really cannot cater to CACC or RACC. Hence we say mutation testing as we do it in this course will not subsume correlated active clause coverage criteria and restricted active clause coverage criteria.

(Refer Slide Time: 21:22)

Mutation subsuming all-defs coverage

- We need strong mutation for mutation testing to subsume all-defs coverage.
- To show that mutation testing subsumes all-defs, we consider only statements that contain variable definitions.
- The mutation applied is to delete such statements.
- Assume statement s_i contains a variable x and m_i is the mutant that deletes s_i .
 - To strongly kill m_i , a test case t must cause (1) mutated statement to be reached, (2) lead to an incorrect state after executing the mutated statement and (3) result in incorrect output.
 - Mutated version of s_i will not assign a value to x hence incorrect state will occur.
 - For final output to be incorrect:
 - If x is an output variable (considered a use of x), t gives an execution of a sub-path from infection to output, covering the def of x .
 - If x is not an output variable, not defining x at s_i results in an



Finally we go back to graph coverage criteria. I would like to tell you that mutation testing subsumes all defs. If you remember what is all defs criteria, all defs is that there must be a path which is def clear from every definition to one possible use. In this case we cannot use weak mutation. So, is we switch back to strong mutation. So, strong mutation means that the output should be different not only reachability and infection, but propagation should also be met. So, what to do we do to all defs condition? We consider every statement in the program that contains definitions of variables.

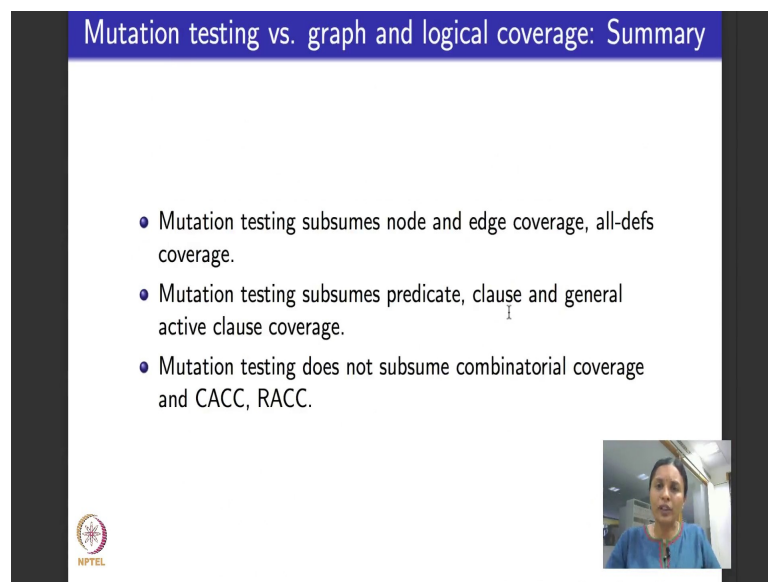
What we do to mutate is that we simply remove that statement, is that clear? We simply remove that statement. When you remove that statement, what happens? Assume that there is a statement s_i that contains the definition of a variable x , and m_i is the mutant that removes or deletes the statement s_i . To strongly kill the mutant program m_i , a test case must cause the following. It must cause the mutated statement to be reached. It must lead to infection, which is, lead to an incorrect state after execution of the mutated statement. And it must lead to propagation, which is result in an incorrect output. Mutated version of s_i will not assign the value to x because s_i itself is not there.

So, the assignment itself is missing. So, an incorrect state will definitely occur, propagation will definitely happen. So, the final output to be incorrect, if this variable x that I am considering is itself an output variable then t gives execution of a sub path from infection to the output variable. If x is not a output variable then not defining x itself

results in an error. I am sorry, I just realize that the last line is missing here, please read it as if x is not an output variable then not defining x as s i results in an error on it is own right. Either way I think we anyway have issues. So, what I am saying is to cover all definitions consider every statement in a program that contains the definition.

The mutation that I apply is, remove that statement, completely remove that statement from the program. And the claim is that if that statement is removed you will definitely meet all the 3 conditions of reachability, infection and propagation. So, the output of the mutated program is guaranteed to be different from the output of the original program because one single statement is missing. Because the statement is missing, mutation testing subsumes all definitions coverage.

(Refer Slide Time: 24:00)



Mutation testing vs. graph and logical coverage: Summary

- Mutation testing subsumes node and edge coverage, all-defs coverage.
- Mutation testing subsumes predicate, clause and general active clause coverage.
- Mutation testing does not subsume combinatorial coverage and CACC, RACC.

The slide includes the NPTEL logo in the bottom left corner and a small video inset in the bottom right corner showing a woman speaking.

So, just to summarize what we learnt today, in the graph coverage criteria, mutation testing subsumes node, edge coverage amongst the structural criteria. And in the data flow criteria, it subsumes all defs coverage. In logical coverage criteria, mutation testing subsumes predicate coverage, clause coverage and generalized inactive coverage. It does not subsume combinatorial coverage, CACC and RACC. It is not known whether mutation testing subsumes inactive clause coverage criteria. It is also not known whether mutation testing subsumes all uses criteria. So, if you pursuing research here are 2 good problems for you to work on. See and understand, and see if you can infer if mutation testing, how does mutation testing relate to inactive clause coverage criteria. And how

does mutation testing relate to all uses or all du paths criteria. So, next week, we will begin with applying mutation testing for design integration. That is it for this week.

Thank you.

Software testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

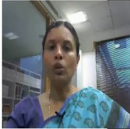

Lecture – 41
Mutation Testing

Welcome to week 9, we continue with mutation testing this week. This will be the first uploaded lecture for week 9 now. What are we going to do today? We will begin by recapping what we have planned to do for mutation testing. If you remember I have showed you this figure last week in one of my lectures, I say this is an overview of what we are going to do for mutation testing.

(Refer Slide Time: 00:21)

Mutation testing for software artifacts: An overview				
	For programs	Integration	Specifications	Input space
BNF grammar	Programming languages	–	Algebraic specifications	Input languages like XML
Summary	Compilers			Input space testing
Mutation	Programs	Programs	FSMs	Input languages like XML
Summary	Mutates programs	Tests integration	Model checking	Error checking

Focus of this lecture and next: Mutation for programs for integration testing.



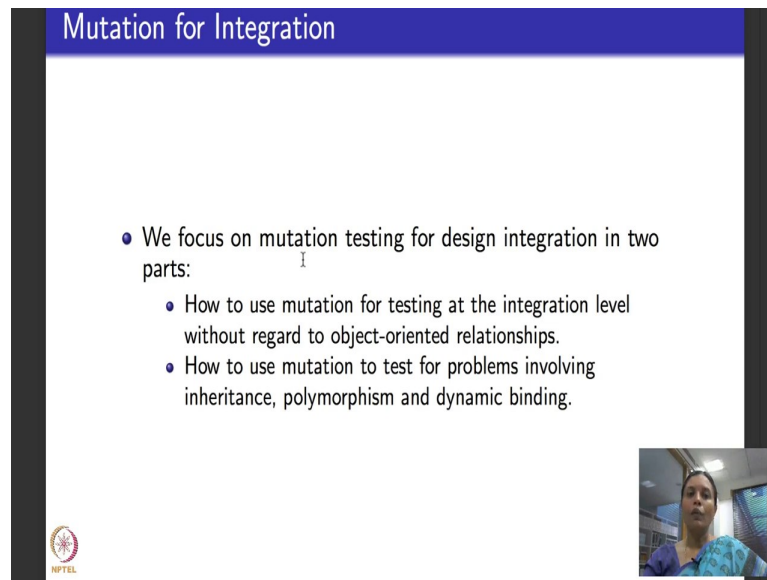
Two kinds of mutation testing, one that directly tests based on the BNF grammar, one that applies the mutation operator and tests that could be applied for programs that unit level for integration testing, for specifications and to mutate the input spaces and in programs you could apply it for any programming language, you could apply to test compilers and you could mutate program and test this is what we had looked last week. There is no known application of BNF grammar based testing for integration testing, but you can apply mutation operators to program to do integration testing. When it comes to specifications there are specific classes at specifications called algebraic specifications

for which mutation testing can be used, you can also apply to finite state machines, you can apply it to model checking tools like SMV and NuSMV.

In input space partitioning the main goal of mutation testing is to apply it to input description languages, we will see it for XML. So, this was an overview of all that we were going to do regard to mutation testing for software artifacts. What have we done already? The things that are colored in green here, we have completed BNF grammar for programming languages. I told you how to mutate by using the grammar, we also saw mutation operators for programs strongly killing, weakly killing mutants, ground strings and we saw an exhaustive list of mutation operators that you could apply for programs written in Java and C to mutate your program and do unit testing for that program. So, these we have already completed. The things that I have stricken out, things that as I told you we are not going to cover as a part of this course. So, testing of compilers is a very involved topic that based that is based on grammar testing, is not within the scope of this course. So, I am not going to do that.

Algebraic specification needs me to introduce the detail notion of algebraic specifications, and it does not have great practical applications. So, I decided to skip that also, and write in the beginning I told you this course does not cover model checking. So, we will not really do mutation operators for model checking. So, the pending ones are the ones that are in black to be done. What we are going to do in this lecture and into a good amount in the next lecture would be, how mutation testing is applied to programs to be able to do design integration testing. This is what I am going to tell you. So, this is the focus of the next couple of lectures including this lecture.

(Refer Slide Time: 03:10)



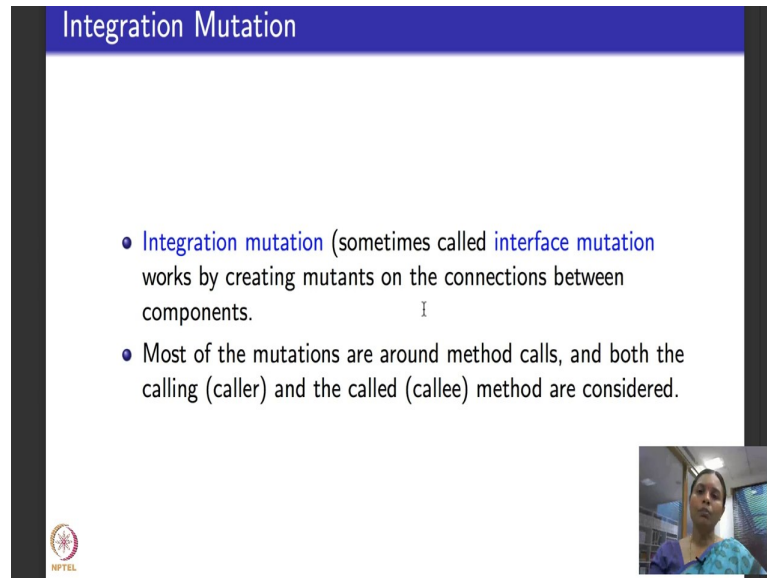
The slide has a blue header with the text "Mutation for Integration". Below the header, there is a bulleted list. The first bullet point is "We focus on mutation testing for design integration in two parts:". The second bullet point is "How to use mutation for testing at the integration level without regard to object-oriented relationships.". The third bullet point is "How to use mutation to test for problems involving inheritance, polymorphism and dynamic binding.". In the bottom right corner of the slide, there is a small video inset showing a person speaking. In the bottom left corner, there is a small logo with the text "NPTEL" below it.

- We focus on mutation testing for design integration in two parts:
 - How to use mutation for testing at the integration level without regard to object-oriented relationships.
 - How to use mutation to test for problems involving inheritance, polymorphism and dynamic binding.

So, when it comes to mutation programs at the level of integration testing, we have considered two parts of it. The first that we ask is how to use mutation for testing at integration level without focusing on object orientated relationships. Let us say I have a C program; the C program can be modular it could have several procedure that call each other, several functions that call each other, and I might want to test the interfaces for the procedure calls related to C. There will be no object oriented features at all, we will do that separately that is such that it applies to any procedures methods, called interfaces and could be used generically cutting across several different programming languages. Then we will consider specifically integration related to object oriented features, because that is very important than lot of new complications come when it comes to integrating modules from different classes modules from different methods.

So, we will look at a separate set of mutation operators that focus on object oriented integration testing. So, I will begin this lecture in this lecture we will do the first part I will tell you what are the inter mutation operators for mutation testing at integration level for generic programming language. You could use it for generic programming language like C, you could also use it for a programming language like Java, but without focusing on the object oriented relationships or any other programming language.

(Refer Slide Time: 04:35)



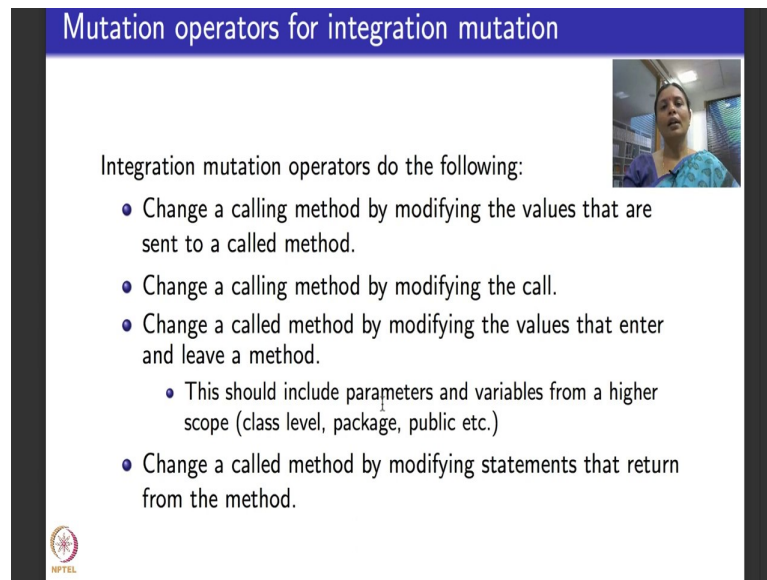
The slide is titled "Integration Mutation" in a blue header bar. It contains two bullet points: "• Integration mutation (sometimes called interface mutation) works by creating mutants on the connections between components." and "• Most of the mutations are around method calls, and both the calling (caller) and the called (callee) method are considered." There is a small video inset in the bottom right corner showing a person speaking. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.

- Integration mutation (sometimes called interface mutation) works by creating mutants on the connections between components.
- Most of the mutations are around method calls, and both the calling (caller) and the called (callee) method are considered.

So, integration mutation, integration testing or integration mutation is also called interface testing. If you remember in the module that I told you about design integration when I do integration testing. The focus is on interfaces also called interface testing, how does it work? It works by creating mutants. Where does it create mutants? It creates mutants on the interfaces or on the connections between the components, it does not mutate it within a method that we already saw last week.

The mutation that we want to focus on this week will focus on mutation on the calls are interfaces that are present between the functions and procedures and methods. Most of the mutations that we will do are around method calls or function calls, both the calling method and the called method are considered for mutations.


(Refer Slide Time: 05:24)



Mutation operators for integration mutation

Integration mutation operators do the following:

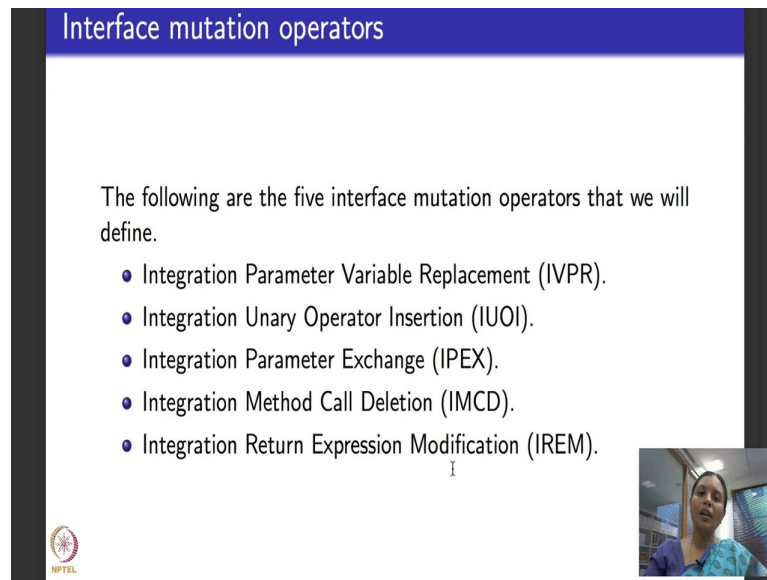
- Change a calling method by modifying the values that are sent to a called method.
- Change a calling method by modifying the call.
- Change a called method by modifying the values that enter and leave a method.
 - This should include parameters and variables from a higher scope (class level, package, public etc.)
- Change a called method by modifying statements that return from the method.



So, generically listing these are what the integration mutation operators do. They first, this is the first to talk about the calling method, the last to talk about the called method. The first one says if they change a calling method by modifying the values that are sent to the called method. So you could change variable names, you could change expressions that are sent to the call method. The second mutation that again focuses on the calling methods, it is that is modifies the call itself, we will see examples of how to do this. The other two kinds of mutation focuses on the method that is being called, called method. It changes the called method by modifying the values that enter and leave the method. What does the called method return, what is out what is it enter, what does it leaves it.

So, typically it includes parameters and variables from highest scope also. For object oriented programming including variable set come from a class level, from package level public variables and so on. Of course, if you are in C it does not make sense, but if you are working with object oriented programs then you need to worry about this also. The last kind of mutation will change the call method by modifying statements that return from the method. So, here it modifies the values that enter and return from the method, it here it modifies the statement itself that return the method, only the return statements. We do not focus on modifying the internal statements when we do unit testing. We would have tested that method separately by modifying the internal statements, we don't have to consider that.

(Refer Slide Time: 06:54)



The following are the five interface mutation operators that we will define.

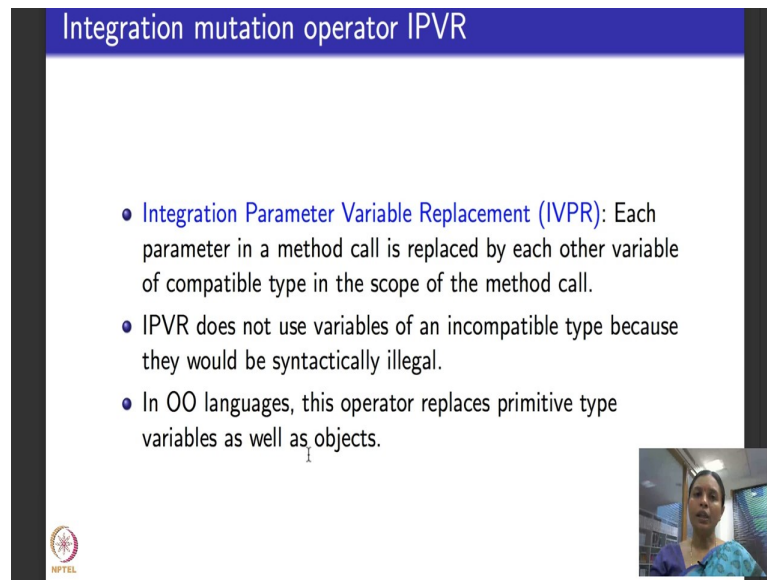
- Integration Parameter Variable Replacement (IVPR).
- Integration Unary Operator Insertion (IUOI).
- Integration Parameter Exchange (IPEX).
- Integration Method Call Deletion (IMCD).
- Integration Return Expression Modification (IREM).

The slide features a blue header with the title 'Interface mutation operators'. Below the header, the text 'The following are the five interface mutation operators that we will define.' is followed by a bulleted list of five operators. In the bottom right corner, there is a small video inset showing a woman speaking. The NPTEL logo is visible in the bottom left corner of the slide.

So, here are five generic interface mutation operators that we will be discussing. As I told you can apply it to any programming language that is modular, without it being object oriented it applies to any programming language. So, we will see them one at a time. The first one we will see is called integration parameter variable replacement. Do not worry too much about these abbreviations, I have given them because the book has them and sometimes it useful as a matter of convenience, but you do not have to remember the abbreviation and they are by no means important for us. So, if you see every operator begins by with the word integration. So, it says what do I do in integration. I replace a parameter variable that is the first mutation, the second one says in integration testing, I reply I insert a new unary operator. If you call a method with x comma y may be instead of calling a method x comma y, as arguments you will call it with minor x comma y that would be a unary operator insertion mutation.

The third one says I change the parameters that are called then doing the interface or integration. The fourth one says I delete the method call itself see what happens. It is a natural thing to do right, now maybe it is a redundant call. So, its mutation could delete the entire method call itself that last mutation says I modify the return expression returned by the called method.

(Refer Slide Time: 08:23)



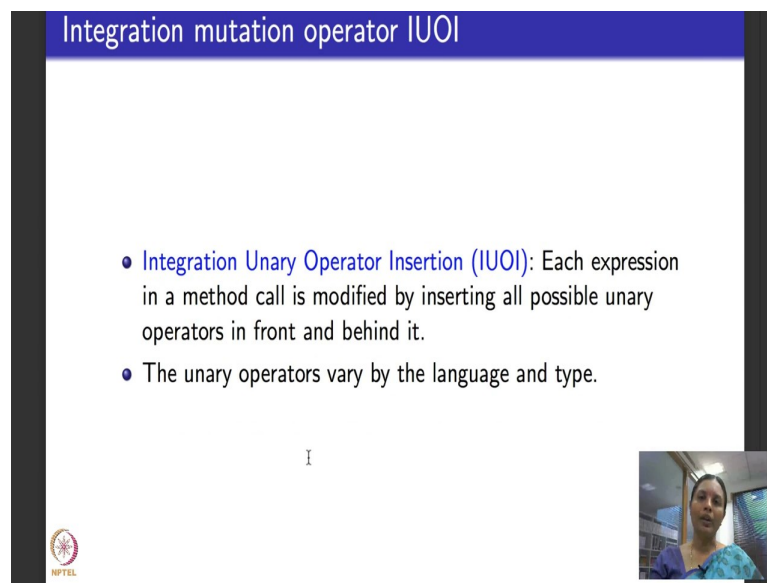
The slide has a blue header with the text "Integration mutation operator IPVR". Below the header, there are three bullet points in blue text. In the bottom right corner of the slide, there is a small rectangular video inset showing a person. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.

- **Integration Parameter Variable Replacement (IPVR):** Each parameter in a method call is replaced by each other variable of compatible type in the scope of the method call.
- IPVR does not use variables of an incompatible type because they would be syntactically illegal.
- In OO languages, this operator replaces primitive type variables as well as objects.

So, we will go ahead and define one at a time, the first one integration mutation operator I V P R expands as integration parameter variable replacement. So, what does it say? If you read it out, it says there is a parameter which are the parameters with which the method is being called by the column method. This mutation says, take each parameter in the method call and replace it with each other variable of compatible type. So, let us say that is a call to a method called g with two parameters a and b, maybe you replace it with c and d such that type of c matches with the type of a the type of b matches with the type of d. Why is type matching important? It is important, the type should be compatible or matching. Remember it is important because the mutated program should be able to compile execute and run for the notion of killing.

So, I cannot replace it with any other variable, I replace the parameters with other variables that are of this same type of compatible type that are also present in the program in object oriented languages. When we apply integration parameter variable replacement, we also have to consider replacing primitive type variables and objects. So, in summary, what is this mutation operator says? It says that the caller is calling a calling procedure with some variables, those variables are called parameters go ahead and change these parameters to any other variable of a compatible type that is all it says.

(Refer Slide Time: 09:51)



The slide has a blue header with the text "Integration mutation operator IUOI". Below the header, there are two bullet points:

- **Integration Unary Operator Insertion (IUOI):** Each expression in a method call is modified by inserting all possible unary operators in front and behind it.
- The unary operators vary by the language and type.

Below the bullet points, there is a small, faint letter 'I'. In the bottom right corner of the slide, there is a small video inset showing a person speaking.

The next mutation operator called as integration unary operator insertion, what is it do? It says there is a method call there is an expression in the method call you modify the expression by inserting all possible unary operators in front and in behind.



So, as I told you suppose there was a call to a method at some point called, let us say call to a method f , which had the three parameters as a , b and c , you could insert a unary operator in front of a to mutate I, t to make it minus a . So, instead of the method being called with parameters a , b and c , it will be called with parameters minus a , b and c . In turn you could mutate and change the sign of b , you could mutate the change and change the sign of c and so on. So, unary operators that are available vary by language and by type. So, whichever language you are using basically whatever is compatible in that language, feel free to pick up that unary operator and change the parameters by inserting a new unary operator in front of a reference variable or a parameter.

(Refer Slide Time: 10:53)

Integration mutation operator IPEX

- **Integration Parameter Exchange (IPEX):** Each parameter in a method call is exchanged with each parameter of compatible type in that method call.
- For e.g., if a method call is `max(a,b)`, a mutated method call of `max(b,a)` is created.

I





The next operation, third operation mutation operation is this, it says integration parameter exchange, which means what each parameter in a method call is exchanged with each parameter of compatible type in that method call. What is it mean, it says if the method call, for example, is something like this a comma b, then I exchange the parameters a and b and instead create a call which says max of b comma a. Maybe this will find a bug, this is a useful interface mutation operator.

(Refer Slide Time: 11:34)

Integration mutation operator IMCD

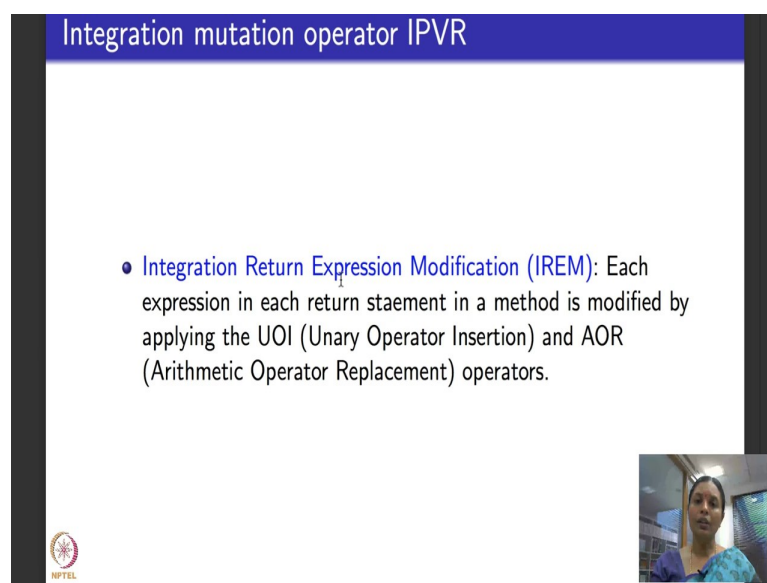
- **Integration Method Call Deletion (IMCD):** Each method call is deleted. If the method returns a value and it is used in an expression, the method call is replaced with an appropriate constant value.
- In Java, the default values should be used for methods that return values of primitive type. If a method returns an object, the method call should be replaced by a call to `new()` on the appropriate class.



The next one that we will see is what is called integrator integration method call deletion. So, what it says as I told you here, it is bold, it just deletes the entire statement that touches the method of the function call. So, if each call is deleted means, what if it is deleted, the mutated program should still compile and produce the value. So, it should be the method is actually originally called would have returned the value. So, when you delete the method call there will be no returning of value, and the program execution not compiling will stop. So, you cannot just delete and hope everything will be fine.

Obviously the program is not going to compile for you to see if there is any error. So, when you delete you also do the following. If the method returns a value and is used in an expression the method call is replaced with appropriate constant value. Let us say there was a method call which was returning an integer value which was used in an expression at a subsequent point in the program in the main calling program. So, what you do? You go ahead and replace wherever it was used. This method call was used with a simple assignment of some constant integer value. So, that the main program continues to compile and execute without any problem. When I specifically apply this method called deletion in programming language like Java I must be careful, the default value should be used for methods that return values of primitive types also. The method returns an object then the method call should be replaced by a call to the special new method within the appropriate class otherwise will run into trouble.

(Refer Slide Time: 13:04)



Integration mutation operator IPVR

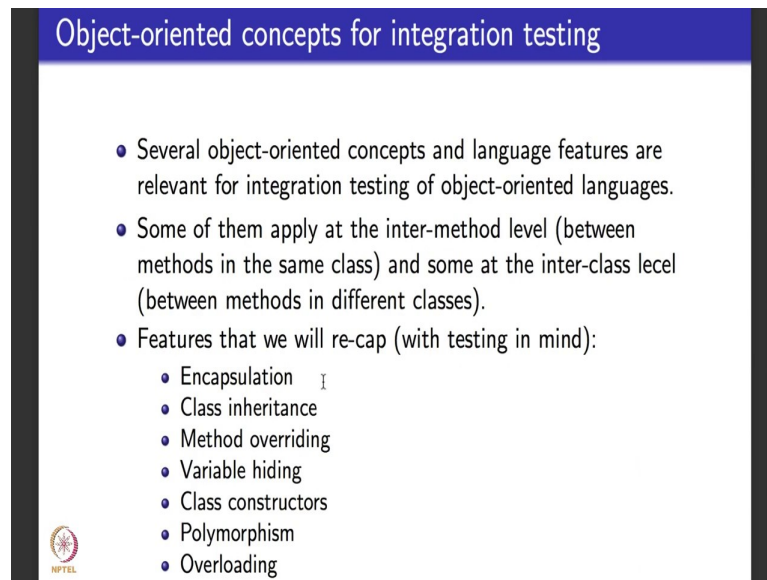
- **Integration Return Expression Modification (IREM):** Each expression in each return statement in a method is modified by applying the UOI (Unary Operator Insertion) and AOR (Arithmetic Operator Replacement) operators.

The slide features a blue header with the title 'Integration mutation operator IPVR'. Below the header, a single bullet point describes the IREM operator. In the bottom right corner, there is a small video inset showing a woman speaking. The NPTEL logo is visible in the bottom left corner of the slide area.

So, the final genetic mutation operator that we will see is what is called integration return expression modification, what does it say? It says each expression in a return statement in a method is modified by using any of these operators. If you remember we had seen these two operators in the last lecture. Last week's lectures, when I had explained to you about method unit level integration operators with same category of operators one after the other, we had specifically seen this unary operator insertion which inserts the unary operator. Similar to the one we saw here in this slide is in such unary operators for parameters, this could insert unary operator in any expression and then there was arithmetic operator replacement which let us you replace with the plus with a minus plus with a star minus with a star, star with a plus and so on. So, you could use any of these mutation operators and change the expression that was involved in the callee method returning the value to the called method. As I told you, do not make any other changes to the internals of the callee method; only the return part of the expression is changed because our focus is only on interfaces.

Just to summarize what I set out by giving our five generic interface mutation operators, which could be applied for any programming language, the first in computation operator says you check and change or replace the parameters that are called during the call, you could change the parameter by inserting a negation or a unary operator you could exchange the parameters, you could delete the method call, but replace it with a dummy stuff so that the main program continuous to run or you could change the return expression modification by using any of the method level mutation operators. All these 5 mutation operators can be applied to any programming language, when you want to focus and test interfaces.

(Refer Slide Time: 15:02)



The slide has a blue header with the title "Object-oriented concepts for integration testing". Below the header, there is a list of bullet points. The first three are at the top level, and the last one is a sub-list under the third bullet point. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.

- Several object-oriented concepts and language features are relevant for integration testing of object-oriented languages.
- Some of them apply at the inter-method level (between methods in the same class) and some at the inter-class level (between methods in different classes).
- Features that we will re-cap (with testing in mind):
 - Encapsulation
 - Class inheritance
 - Method overriding
 - Variable hiding
 - Class constructors
 - Polymorphism
 - Overloading

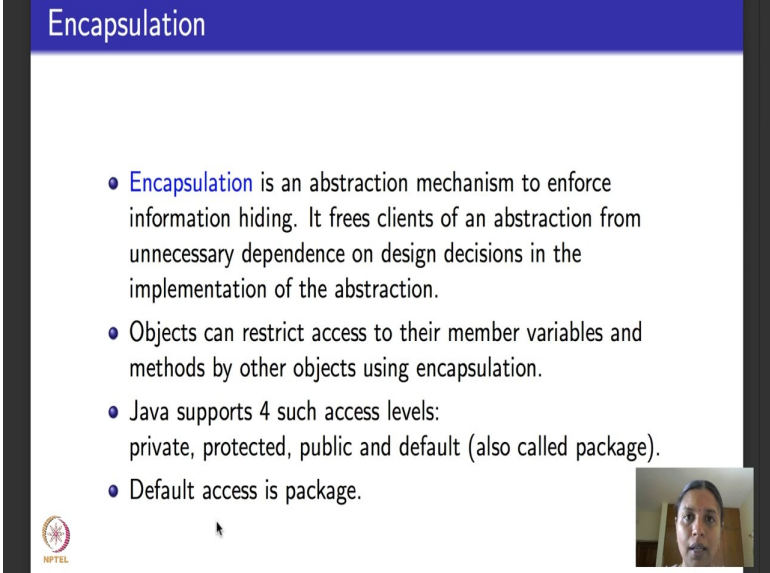
So, towards moving onto using mutation testing specifically for object oriented languages, first point to be noted as I told you is that for object oriented languages the kind of integration that happens is very different from imperative languages like C and all that. So, it helps to spend some time understanding what are the various concepts underlying object oriented programming language that helps us to integrate software components that are developed for these languages. So, with that in mind I would like to recap from the point of view of testing the following object oriented concepts, and before we move on and do that.

When it comes to integration testing for object oriented languages we should remember that there could be two levels, one is you have developed individual methods and you want to put together the methods in the same class and test how they interact. The other is you have written more code you developed a lot of code for several different classes and you want to put together the code of these classes, and test the interactions between the methods that come from different classes. Whatever it is the concept that we will be recollecting today, will help us to do both these kinds of integration tests.

So, what are the features that we will recap? Please note that this not meant to be an exhaustive introduction to these features, we are only going to give a brief overview of the listed features here from the point of view of our use in testing. So, will use it from here for mutation testing, and later when I do exclusive modules and object oriented

testing we will revisit some of these features. So, I will tell you what encapsulation is, class inheritance, method overloading method overriding, hiding of variables class constructor and the notion of polymorphism.

(Refer Slide Time: 16:56)



The slide is titled "Encapsulation" in a blue header. It contains a bulleted list of four points. In the bottom right corner, there is a small video inset showing a person's face. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.



- **Encapsulation** is an abstraction mechanism to enforce information hiding. It frees clients of an abstraction from unnecessary dependence on design decisions in the implementation of the abstraction.
- Objects can restrict access to their member variables and methods by other objects using encapsulation.
- Java supports 4 such access levels: private, protected, public and default (also called package).
- Default access is package.

So, the first one in our list is encapsulation, what is encapsulation? Remember the center of object oriented languages is to do abstraction. Encapsulation is one of the extraction mechanisms, that several object oriented languages offer. Encapsulation and abstraction that enforces information hiding; the word encapsulate says you close something you hide something. It frees the clients of an abstraction from unnecessary dependence on some design decisions that you do not want to happen at that level of abstraction. Sometimes at an abstract level you want to not know what certain decisions and a refined level of the abstraction you want to know about the design decisions. Encapsulation helps you to do that. True encapsulation objects can restrict that access to member variables and methods by other objects using encapsulation.

Java support four different access levels, other programming languages that support object oriented features might have other access levels. The four access levels that we will discuss with reference to Java in mind are private, protected, public and default. Default is also called package because default is also called package, the default access if not specified is package.

(Refer Slide Time: 18:13)

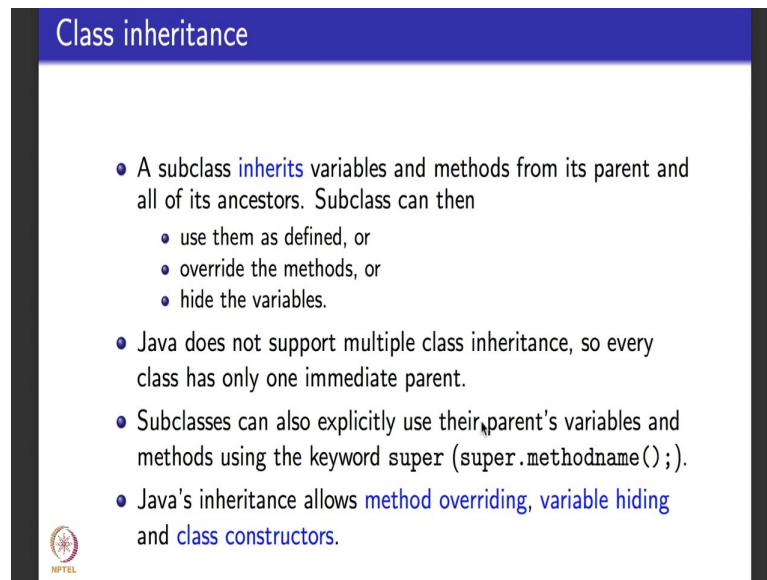
Encapsulation: Access levels in Java				
Specifier	Same class	Different class same package	Different package subclass	Different package non-subclass
private	Y	n	n	n
package	Y	Y	n	n
protected	Y	Y	Y	n
public	Y	Y	Y	Y



So, how do these access levels look like? For example, if a particular entity is private let us say a variable is private, then here are all the access level read capital, yes capital Y as yes, n as, no small n as no. So, if a particular variable is private then within the same class is available for access, from different class same package in different package subclass in different package, in a different class it is not available. If a variable is declared as package, which is the default declaration is available within the same class, in different class, but within the same package, but it is not available in a different package, in different a subclass variable.


If a variable is protected then its yes for class, yes for different class and same package, yes for different package and no for different package non subclass public access is available through out there is no restriction the variable or entity can be seen everywhere. So, this is what encapsulation deals with.

(Refer Slide Time: 19:15)



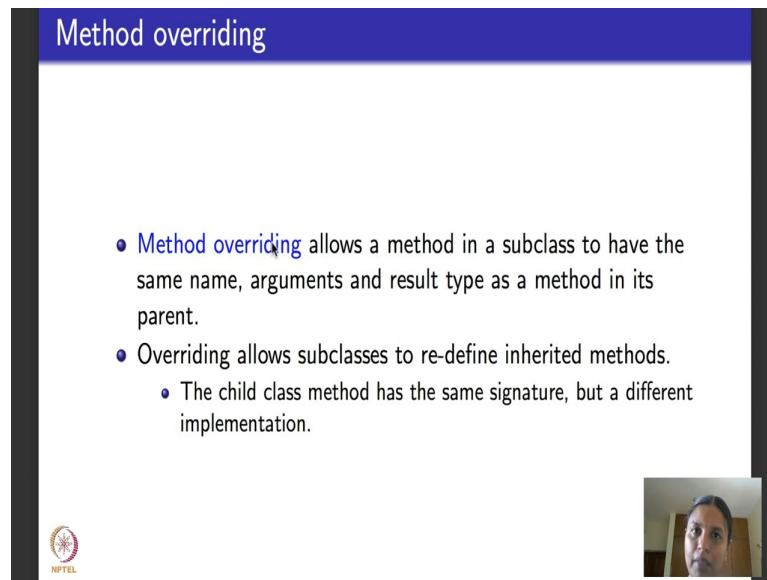
Class inheritance

- A subclass **inherits** variables and methods from its parent and all of its ancestors. Subclass can then
 - use them as defined, or
 - override the methods, or
 - hide the variables.
- Java does not support multiple class inheritance, so every class has only one immediate parent.
- Subclasses can also explicitly use their parent's variables and methods using the keyword `super` (`super.methodname();`).
- Java's inheritance allows **method overriding**, **variable hiding** and **class constructors**.



The next concept that we want to look at is what is called class inheritance. What is class inheritance? Let us talk about. We say one class inherits from another class then the class that it inherits from is called a parent class, and the other class is called a subclass or a child class. Parent of parent could be an ancestor. So, we see a subclass inherits. What are the things that it can inherit? It can inherit variable, it can inherit methods, it can inherit these two entities, variables and entities from its methods parent, or it can inherit from its parent's parent which will be an ancestor and moving on parent's parent's parent, which will be another ancestor and so on. Then after inheriting what can the sub class do with the variable and methods? It can use them as they are defined, it can overwrite the methods redo them or it can hide the methods. We will see overriding and hiding very soon now. Java does not support multiple class inheritance. So, every class has only one immediate parent. Now the sub classes that inherit can explicitly use their parent variables and methods without changing or overriding by using the word `super`. So, the prefix whatever variable or method name with `super`, like this, `super dot method name` then you can use it as it is.


(Refer Slide Time: 20:46)



Method overriding

- **Method overriding** allows a method in a subclass to have the same name, arguments and result type as a method in its parent.
- Overriding allows subclasses to re-define inherited methods.
 - The child class method has the same signature, but a different implementation.

NPTEL





Java's inheritance allows us to do, as I told you here, override methods, hide the variables can also use class constructors. So, we will look at method overriding first. What does method overriding do? Method overriding allows a method in a subclass to have the same name, same arguments and same result type as a method in its parents, but inside, the code that the method implements could be different. So, for all practical purposes its name is also same, arguments is also same, written type is also same, but whatever it computes or whatever is core functionality is, that could be different. Overriding what does it do? It allows subclasses to redefine inherited methods; child class method can have the same signature meaning same name arguments and results, but a different implementation that is what I told you just now.

(Refer Slide Time: 21:27)

Variable hiding

- **Variable hiding** is achieved by defining a variable in a child class that has the same name and type of an inherited variable.
- This has the effect of hiding the inherited variable from the child class.




Variable hiding: what is that? Variable hiding is achieved by defining a variable in a child class that has the same name and the type of an inherited variable. So, which means what? You have a variable from your parent class, you declare another variable with exactly the same name, with exactly the same type. So, it is as good as saying I am not considering the variable that I have inherited from the parent class or in other words I am hiding the variable that I have inherited from the parent class.

(Refer Slide Time: 21:58)

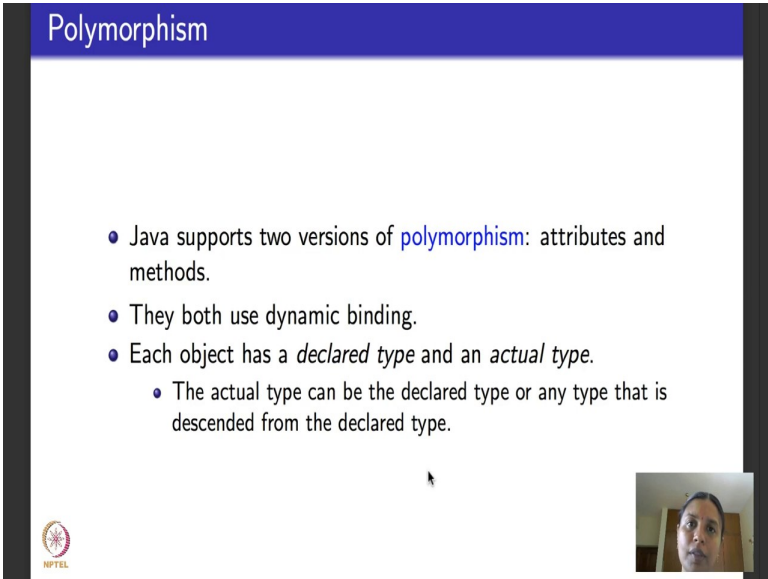
Class constructors

- A **constructor** in a class is a special type of subroutine called to create an object.
- It prepares the new object for use, often accepting arguments that the constructor uses to set required member variables.
- They are not inherited in the same way methods are.
- To use a constructor, we must explicitly call it using the **super** keyword. The call must be the first statement in the derived class constructor and the parameter list must match the parameters in the argument list of the parent constructor.



What is the class constructor? A class constructor is a class in a class, it is a special type of subroutine that is called to create an object. It prepares the new object for use of accepting arguments that the constructor uses to set the required member variables of that object that is being created. These objects are not inherited the same way methods are constructors are very specific features; to use a constructor is explicitly call it again using the super keyword that we saw here using this super keyword.

(Refer Slide Time: 22:41)

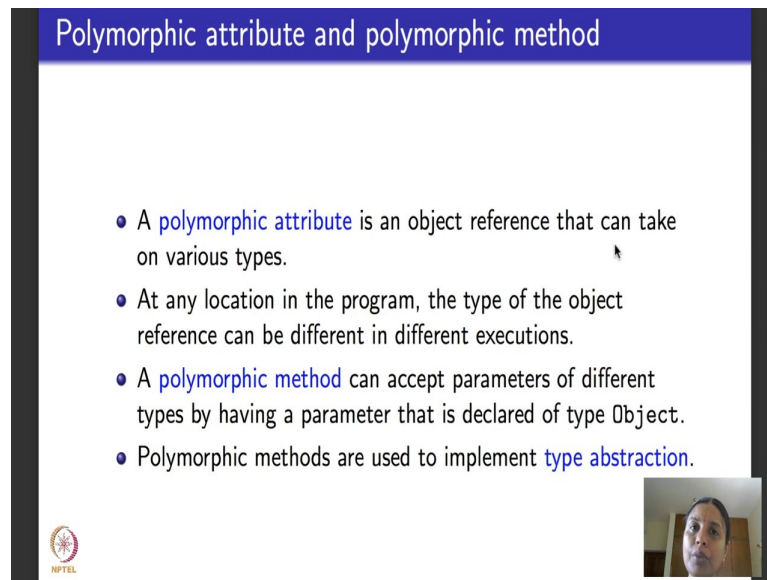


The slide is titled "Polymorphism" in a blue header. It contains a bulleted list of points about Java's polymorphism. In the bottom right corner, there is a small video inset showing a person's face. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.

- Java supports two versions of **polymorphism**: attributes and methods.
- They both use dynamic binding.
- Each object has a *declared type* and an *actual type*.
 - The actual type can be the declared type or any type that is descended from the declared type.

The call must be the first statement that derived class constructor and the parameter list must exactly match the parameters in the argument list of the parent constructor. What is polymorphism? Java supports two versions of polymorphism: attributes and methods both of them use dynamic binding. So, what did you mean by that? An object actually in Java has a declared type and actual type. Actual type can be the declared type or it can be any type that is descended from the declared type polymorphic attributes and polymorphic methods as I told you here right two types of polymorphism attributes and methods.


(Refer Slide Time: 23:09)



Polymorphic attribute and polymorphic method

- A **polymorphic attribute** is an object reference that can take on various types.
- At any location in the program, the type of the object reference can be different in different executions.
- A **polymorphic method** can accept parameters of different types by having a parameter that is declared of type Object.
- Polymorphic methods are used to implement **type abstraction**.

NPTEL



So, what do they do? Polymorphic attribute is an object reference that can take on various types that is why the word poly, poly means it can be of several types at any point and time in a given piece of code.



At any location in a program the type of object reference can be different in different executions of the same program for methods. The method is called polymorphic, if it can accept parameters of different types, having a parameter that is declared of type object. What are polymorphic methods used for? For polymorphic methods are used to implement type abstraction please remember that is also something called overriding methods which is basically another method in a child class that has the same name argument and type this is different the polymorphic method can accept parameters of different types by having a parameter that is declared of type object these two are different.

The next concept in our list is overloading, overloading is the use of same name for different constructors or methods in the same class.

(Refer Slide Time: 24:08)

Overloading

- **Overloading** is the use of the same name for different constructors or methods in the same class.
- They must have different **signatures**, or lists of arguments.
- Note: Overloading is different from overriding.
 - The former occurs with two methods in the same class, whereas overriding occurs between a class and one of its descendants.





So, the it again intuitively in the English sense of the term overloading, we use the same name again and again. So, you over load the name they must have different signatures or lists of argument unlike overriding. Overloading is different from overriding why overloading a class with two methods in the same class whereas, override occurs between a class and one of its descendants.

(Refer Slide Time: 24:32)

Instance and class variables in Java

- **Instance** and **class** variables are associated with a class.
- Class variables are also called **static**, in the sense that there is only one copy of the variable that is shared with all instances of that class.
- Instance variables belong to an object (an instance of a class). Every instance of that class has its own copy of the variable. Changes made to the instance variable don't reflect in other instances of that class.
- Levels:
 - Instance variables are declared at class level.
 - Class variables are declared with **static**.
 - Local variables are declared within methods.



The next concept is instance and class variables, instance and class variables are associated with a class. Class variables in Java called static variables in the sense that

there is only one copy of the variable that is shared with all instances of that class. Unlike class variables that are static, instance variables belong to an object and instance of a class, every instance can have its own copy of the variable that way it is not static. Changes made to a particular instance of an instance variable do not reflect in other instances of that class. So, instance variables are typically declared at class levels, class variables are also declared at class level, but with the keyword static. Local variables are declared within methods.

So, this was a brief overview of some object oriented concepts that I wanted to tell you. So, we recapped all these concepts that are listed in the slides. This by no means is meant to be an exhaustive introduction. So, please do not consider this as an exhaustive introduction, we just briefly recap concepts as we wanted for testing specifically mutation testing related to object oriented integration. Later when we do object oriented applications testing also will come and recap these concepts feel free to pick up any good book on object oriented programming, and you will get a more detailed thorough introduction to these concepts.

I hope this brief overview was helpful, we will continue with object oriented mutation testing for integration in the next lecture.

Thank you.

Software testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 42
Mutation Testing



Hello there, we are in week 9. Last lecture I did mutation testing for integration specifically focusing on design integration.

(Refer Slide Time: 00:21)

Mutation testing for software artifacts: An overview

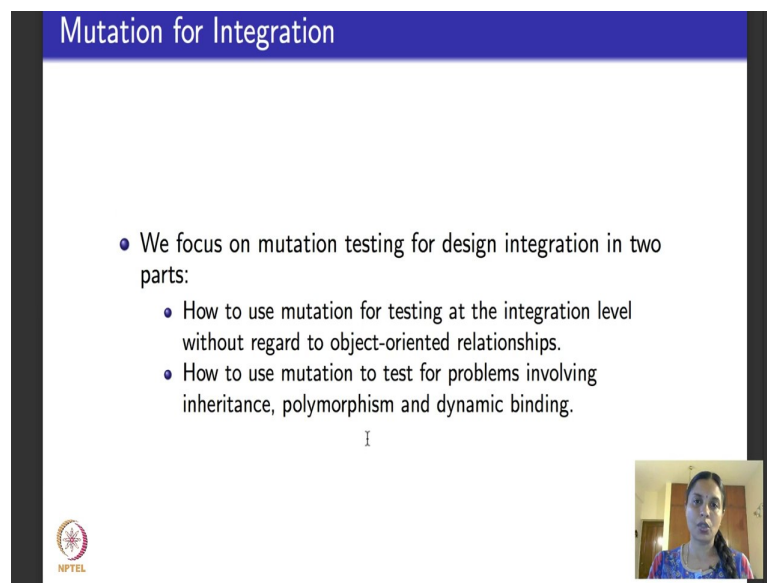
	For programs	Integration	Specifications	Input space
BNF grammar	Programming languages	–	Algebraic specifications	Input languages like XML
Summary	Compilers			Input space testing
Mutation	Programs	Programs	FSMs	Input languages like XML
Summary	Mutates programs	Tests integration	Model checking	Error checking

Focus of this lecture and next: Mutation for programs for integration testing.



So, this is what is the status of our lectures, we are inside mutation testing the ones marked in green are done, the one striked out we are not going to do, this we will be doing in the next two lectures. We are currently here, we are currently here in applying mutation testing for testing program with focus on integration.

(Refer Slide Time: 00:42)



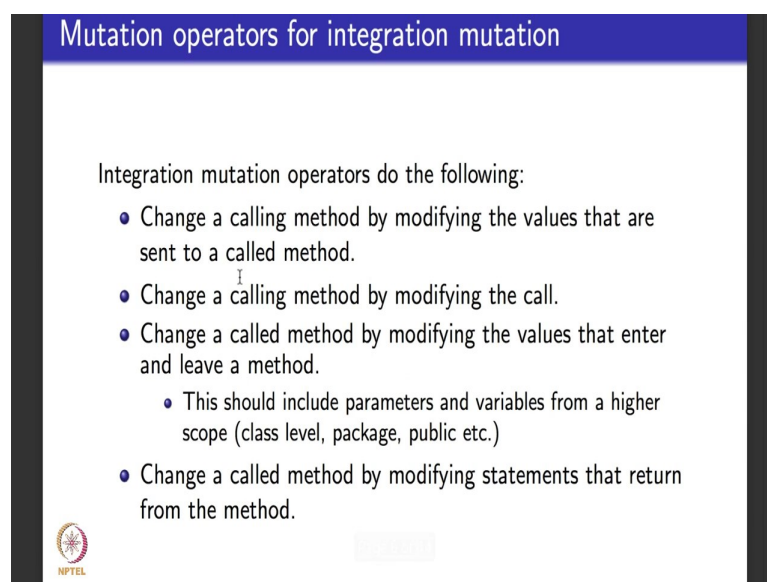
Mutation for Integration

- We focus on mutation testing for design integration in two parts:
 - How to use mutation for testing at the integration level without regard to object-oriented relationships.
 - How to use mutation to test for problems involving inheritance, polymorphism and dynamic binding.

NPTEL

In the first part of this which I taught in the last lecture, I introduced you to generic mutation operators which talked about changing a calling method in more than one ways, change a called method and more than one ways.

(Refer Slide Time: 00:49)



Mutation operators for integration mutation

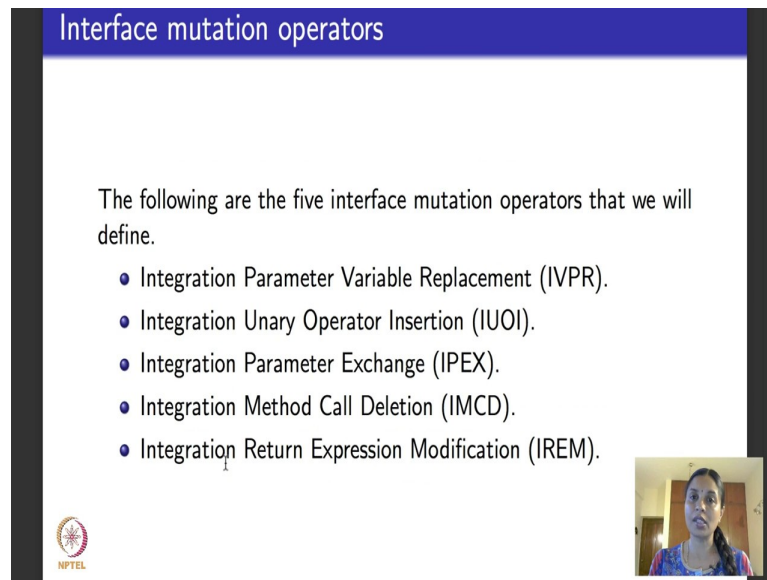
Integration mutation operators do the following:

- Change a calling method by modifying the values that are sent to a called method.
- Change a calling method by modifying the call.
- Change a called method by modifying the values that enter and leave a method.
 - This should include parameters and variables from a higher scope (class level, package, public etc.)
- Change a called method by modifying statements that return from the method.

NPTEL

We discussed these five different generic mutation operators that could be done for integration testing for almost every programming language that supports procedure calls or method calls.



(Refer Slide Time: 00:56)



Interface mutation operators

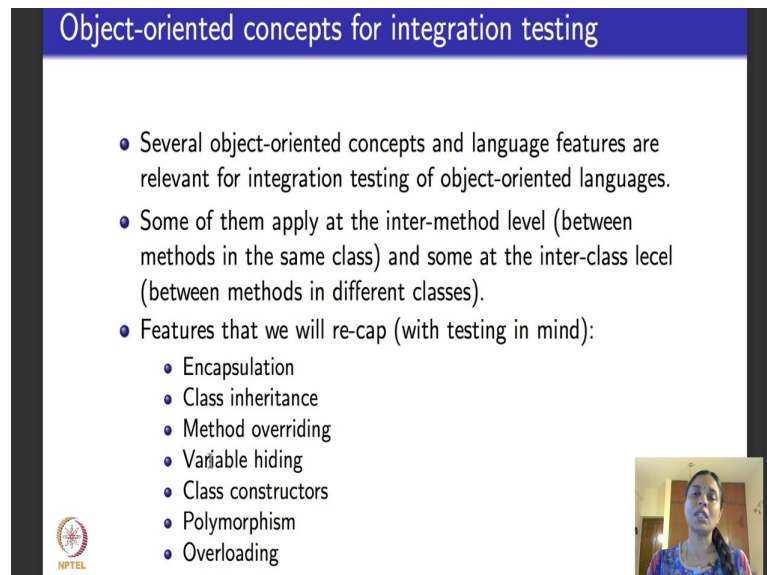
The following are the five interface mutation operators that we will define.

- Integration Parameter Variable Replacement (IVPR).
- Integration Unary Operator Insertion (IUOI).
- Integration Parameter Exchange (IPEX).
- Integration Method Call Deletion (IMCD).
- Integration Return Expression Modification (IREM).





Later on, in the last lecture after doing this what I told you was we said we will focus on object oriented integration testing because object oriented languages have picked up. Lot of web applications, enterprise applications are written in object oriented programming languages.

(Refer Slide Time: 01:11)



Object-oriented concepts for integration testing

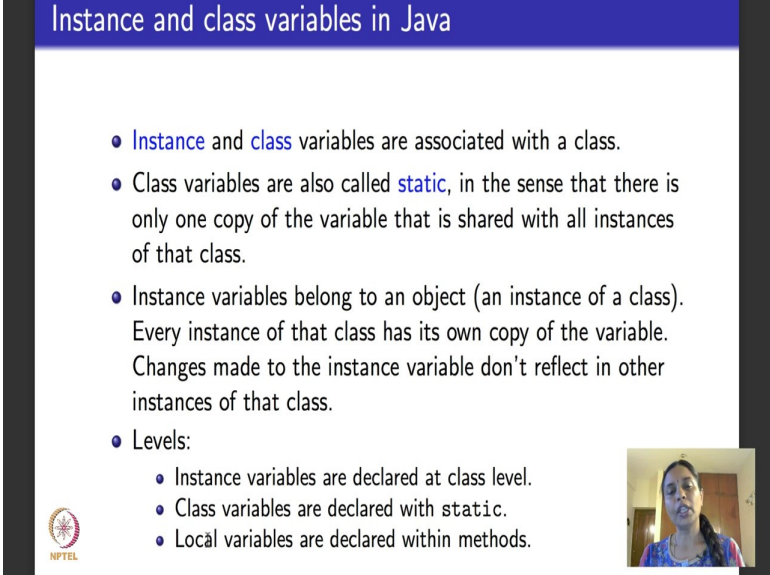
- Several object-oriented concepts and language features are relevant for integration testing of object-oriented languages.
- Some of them apply at the inter-method level (between methods in the same class) and some at the inter-class level (between methods in different classes).
- Features that we will re-cap (with testing in mind):
 - Encapsulation
 - Class inheritance
 - Method overriding
 - Variable hiding
 - Class constructors
 - Polymorphism
 - Overloading



So, we said will focus specifically on mutation operators that are available for object oriented programming languages especially Java. So, with that towards recapping what we need to understand mutation, we recap-ed the essential object oriented features in the

end of last lecture. So, I helped you to recap encapsulation, class inheritance, overriding methods, variable hiding, constructors, polymorphism and overloading.

(Refer Slide Time: 01:59)



The slide is titled "Instance and class variables in Java". It contains a bulleted list of points:

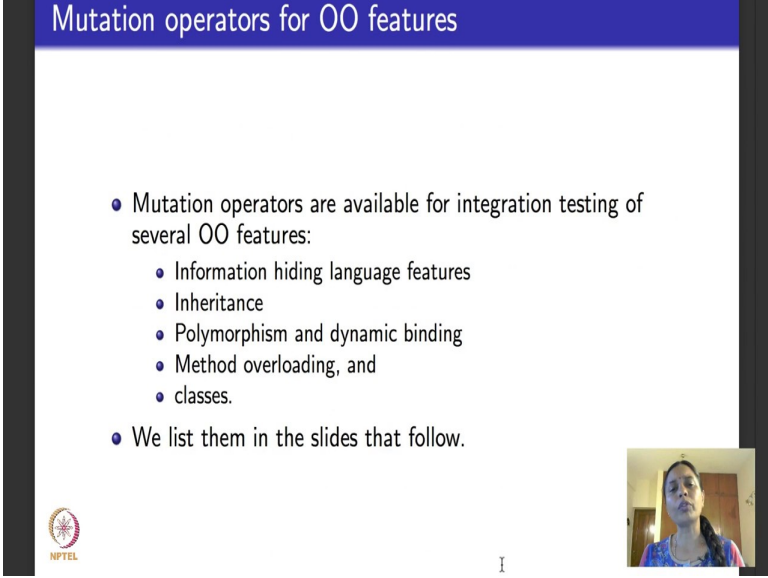
- Instance and class variables are associated with a class.
- Class variables are also called **static**, in the sense that there is only one copy of the variable that is shared with all instances of that class.
- Instance variables belong to an object (an instance of a class). Every instance of that class has its own copy of the variable. Changes made to the instance variable don't reflect in other instances of that class.
- Levels:
 - Instance variables are declared at class level.
 - Class variables are declared with **static**.
 - Local variables are declared within methods.

In the bottom left corner, there is an NPTEL logo. In the bottom right corner, there is a small video inset showing a woman speaking.

So, we did this last time. So, I will skip through these slides. We will directly move into looking at Java and what are the specific mutation operators that are available for focusing and doing integration testing for a language like Java. So, to do that one last concept that I would like to specifically recap related to Java is the notion of instance variables and class variables. So, instance and class variables, both are variables that are associated with a class in Java.

What is the difference, we will see the difference class variables are also called static variables, they are declared using the word static. They are static in the sense that there is only one copy of the variable that is shared with all the instances of the class. Instance variables on the other hand, they belong to an object or an instance of a class and every instance of the class has its own copy of the variable. Changes made to an instance variable do not reflect, suppose you have one copy of instance variable in one object, you make a change it does not reflect on all the other copies of instance variable. Like we had access levels for encapsulation instance variables where are they declared they are declared at class level. Class variables are declared with a keyword static as I told you and in addition to these two we also have local variables which are declared within methods in a class.

(Refer Slide Time: 03:13)



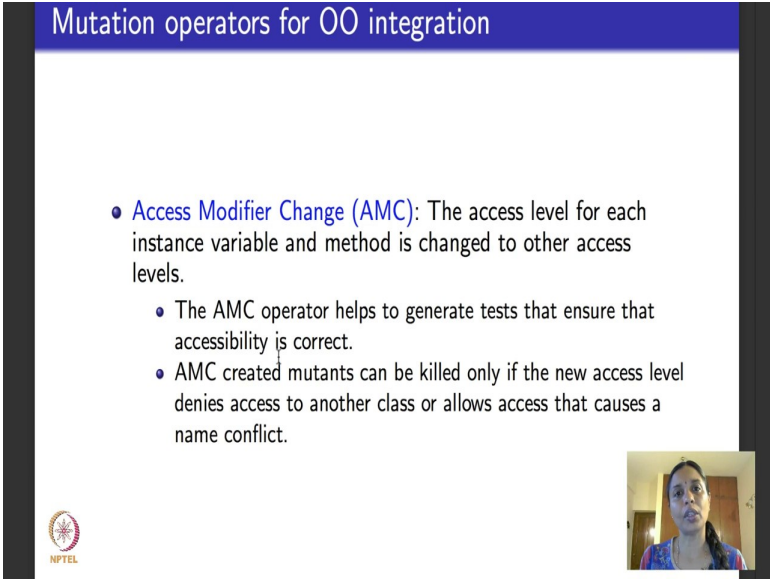
The slide has a blue header with the text "Mutation operators for OO features". Below the header, there is a bulleted list. The first bullet point states that mutation operators are available for integration testing of several OO features, followed by a sub-list of five features: Information hiding language features, Inheritance, Polymorphism and dynamic binding, Method overloading, and classes. The second main bullet point states that these features are listed in the following slides. In the bottom left corner, there is an NPTEL logo. In the bottom right corner, there is a small video inset showing a woman speaking.

- Mutation operators are available for integration testing of several OO features:
 - Information hiding language features
 - Inheritance
 - Polymorphism and dynamic binding
 - Method overloading, and
 - classes.
- We list them in the slides that follow.

Now what we are going to see we are going to see mutation operators for doing integrations testing specific to object oriented features. Without loss of generality you can assume that we will be focusing on mutation operators for Java, but equally well applies to any other object for oriented programming that shares the same kind of object oriented features for integrating different pieces of code into one large code. So, the following are the generic features that we will be seeing. We will instantiate them with reference to Java, but let us say you see C++ and then it has this feature you could write you could as well use these mutation operators to be able to do integration testing for C++.

So, we will see information hiding related features, we will see inheritance related mutation operators, we will see mutation operator is related to polymorphism, dynamic binding, method overloading and classes. So, in the slides that follow, I will be listing about 20 different mutation operators. All of them are meant for testing integration of object oriented programs.

(Refer Slide Time: 04:21)



The slide has a blue header with the text "Mutation operators for OO integration". Below the header, there is a bulleted list. The first bullet point is "Access Modifier Change (AMC): The access level for each instance variable and method is changed to other access levels." followed by two sub-bullets: "The AMC operator helps to generate tests that ensure that accessibility is correct." and "AMC created mutants can be killed only if the new access level denies access to another class or allows access that causes a name conflict." In the bottom right corner of the slide, there is a small video inset showing a woman speaking. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.

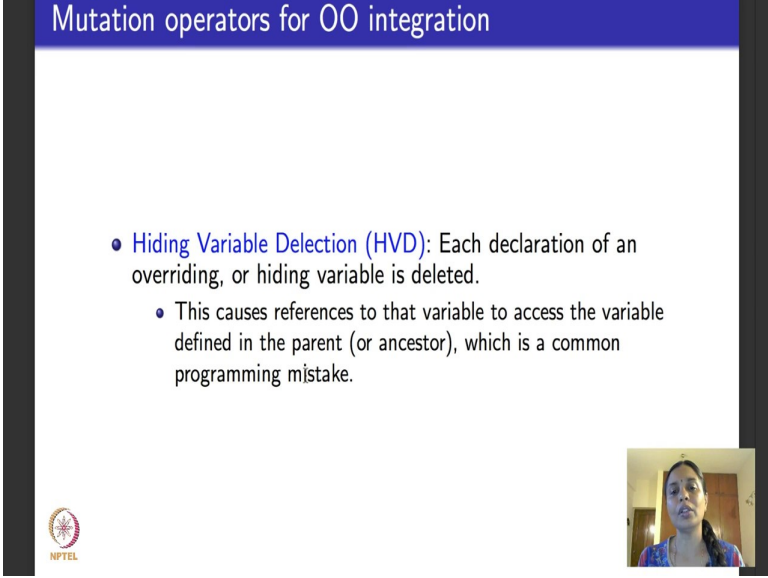
- **Access Modifier Change (AMC):** The access level for each instance variable and method is changed to other access levels.
 - The AMC operator helps to generate tests that ensure that accessibility is correct.
 - AMC created mutants can be killed only if the new access level denies access to another class or allows access that causes a name conflict.

So, for each of these operators, I will tell you what it does and then wherever applicable I will also give you one instance of what is the kind of error that it could be useful to detect while doing design integration. Obviously, when you test your program for design integration you are not going to be able to use all of these operators, this is meant to be used as an exhaustive list. What we are going to use is a select set, as and when we need based on what our focus for integration testing is.

So, the first operator that I am going to tell you is called access modifier change abbreviated as AMC. Do not worry too much about remembering these abbreviations. I just carry them forward because it is useful to illustrate them as you are going on. We really do not need to remember them for any reason at all. What is this operator do? This operator basically changes the access levels. What are the access levels that it changes, it changes the access levels for each instance variable and method.

What do we achieve by this? We achieve and get tests that ensure that accessibility is correctly done during integration. The access modifier change operator creates mutants that can be killed only if the new access level denies access to other classes or it is the opposite, it allows access to other classes which causes some kind of conflict in 'a'. That is when you can observe a change in the behaviour and then you say applying this mutation operator killed the mutant.

(Refer Slide Time: 05:50)



The slide has a blue header with the text "Mutation operators for OO integration". Below the header, there is a bulleted list. The first bullet point is "Hiding Variable Deletion (HVD): Each declaration of an overriding, or hiding variable is deleted." The second bullet point is "This causes references to that variable to access the variable defined in the parent (or ancestor), which is a common programming mistake." In the bottom left corner, there is a circular logo with a star and the text "NPTEL". In the bottom right corner, there is a small video inset showing a woman speaking.

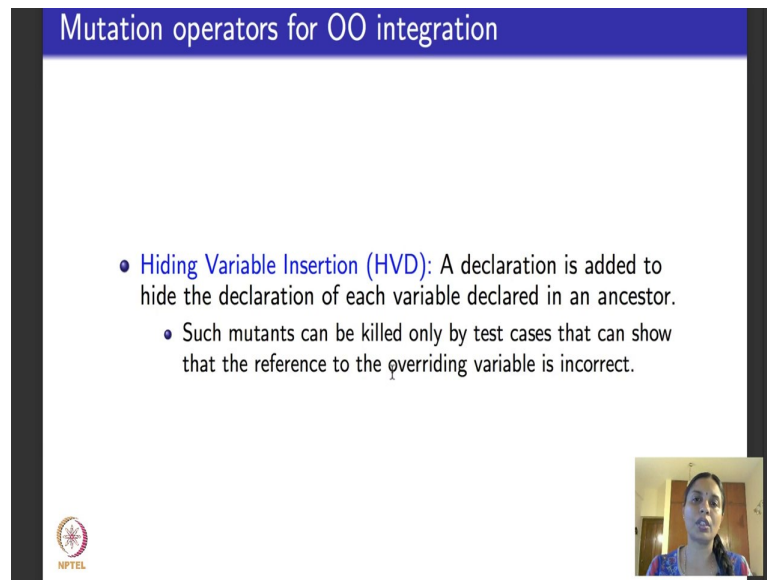
- **Hiding Variable Deletion (HVD):** Each declaration of an overriding, or hiding variable is deleted.
 - This causes references to that variable to access the variable defined in the parent (or ancestor), which is a common programming mistake.

So, the next mutation operator, in fact, all the mutation operators that we will be seeing will only do three kinds or four kinds of changes they will delete something, they will remove something, they will insert something, they will modify something.

So, the first one that we saw was a modification, the next one that we are going to see is deletion. What is this delete? This deletes what are called overriding or hiding variables. The operator is called hiding variable deletion. By just reading the name you should be able to figure out what it is supposed to do it cannot be too difficult. What is this mutation operator do? By hiding or I mean by deleting the overriding or hiding variable we cause references to that variable to access the variable defined in a parent or in any ancestor this could be a common programming mistake that can be caught during integration testing.

The next mutation operator that I am going to tell you specific to Java is an insert mutation operator.

(Refer Slide Time: 06:41)

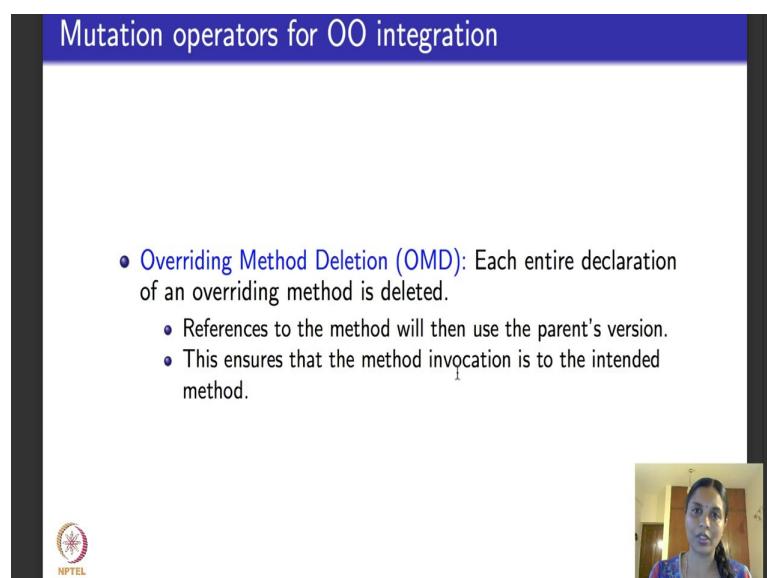


The slide has a blue header with the text "Mutation operators for OO integration". The main content area is white and contains a bulleted list. The first bullet point is "Hiding Variable Insertion (HVD): A declaration is added to hide the declaration of each variable declared in an ancestor." The second bullet point is "Such mutants can be killed only by test cases that can show that the reference to the overriding variable is incorrect." In the bottom left corner, there is a small circular logo with a star and the text "NPTEL". In the bottom right corner, there is a small video inset showing a woman speaking.

- **Hiding Variable Insertion (HVD):** A declaration is added to hide the declaration of each variable declared in an ancestor.
 - Such mutants can be killed only by test cases that can show that the reference to the overriding variable is incorrect.

In the previous one we showed you about deleting hiding or overriding variables. Here what we are going to do is, we add a new declaration. What are we going to do, what are the declaration do that declaration has got to do with hiding variables. So, declaration is added to hide the declaration of each variable declared in an ancestor. What does this mutant do? This mutant basically hides the variables that are declared in the ancestor and it tests whether the overriding reference variable, variable reference is correct or not. So, these mutants can be killed by test cases that show that the reference to the overriding variable is not correct.

(Refer Slide Time: 07:24)



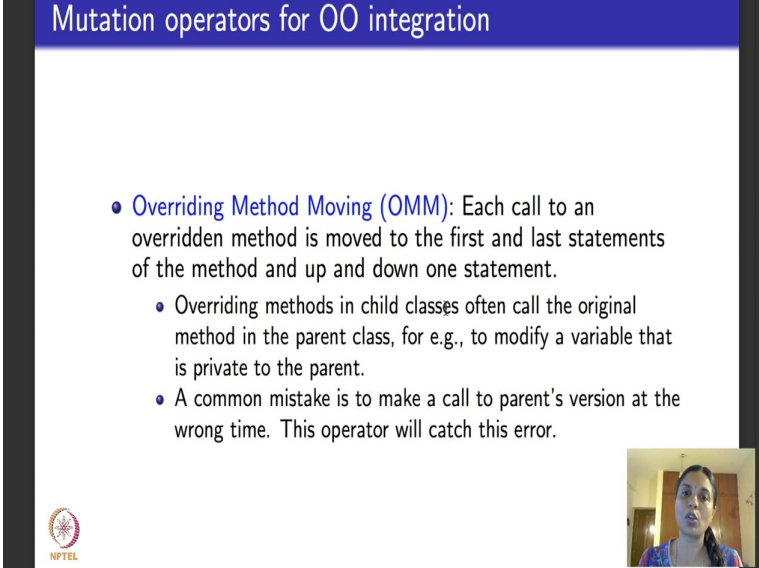
The slide has a blue header with the text "Mutation operators for OO integration". The main content area is white and contains a bulleted list. The first bullet point is "Overriding Method Deletion (OMD): Each entire declaration of an overriding method is deleted." The second bullet point is "References to the method will then use the parent's version." The third bullet point is "This ensures that the method invocation is to the intended method." In the bottom left corner, there is a small circular logo with a star and the text "NPTEL". In the bottom right corner, there is a small video inset showing a woman speaking.

- **Overriding Method Deletion (OMD):** Each entire declaration of an overriding method is deleted.
 - References to the method will then use the parent's version.
 - This ensures that the method invocation is to the intended method.

So, the next mutation operator in our long list of mutation operators that we are going to see, what does it do, it is a deletion mutation operator. Instead of working with variables it works with methods. It is called overriding method deletion abbreviated as OMD, what does it do? It removes the entire declaration of the overriding method. Please note we are not removing the method, we are removing the declaration of the method.

What will then happen? The references to this method that happen in a class will then use the parent's version right. So, if this happens then it ensures that the method invocation is indeed to the correct method and not to the wrong one. By deleting if there is a problem you will be able to find out because the method itself is deleted. So, if there was a correct reference there will be an issue, you will be able to kill the mutant and observe an error. But if there was an incorrect method, by deleting this method reference deleting the method declaration you will not be able to observe a change so that means, you will not be able to kill a mutant. The fact that you cannot kill a mutant reveals an error in this case.

(Refer Slide Time: 08:29)



The slide is titled "Mutation operators for OO integration". It contains a bulleted list describing the Overriding Method Moving (OMM) operator. The first bullet point states that each call to an overridden method is moved to the first and last statements of the method and up and down one statement. The second bullet point explains that overriding methods in child classes often call the original method in the parent class, for example, to modify a variable that is private to the parent. The third bullet point notes that a common mistake is to make a call to the parent's version at the wrong time, and that this operator will catch this error. In the bottom right corner, there is a small video inset showing a woman speaking. The NPTEL logo is visible in the bottom left corner of the slide.

- **Overriding Method Moving (OMM):** Each call to an overridden method is moved to the first and last statements of the method and up and down one statement.
 - Overriding methods in child classes often call the original method in the parent class, for e.g., to modify a variable that is private to the parent.
 - A common mistake is to make a call to parent's version at the wrong time. This operator will catch this error.

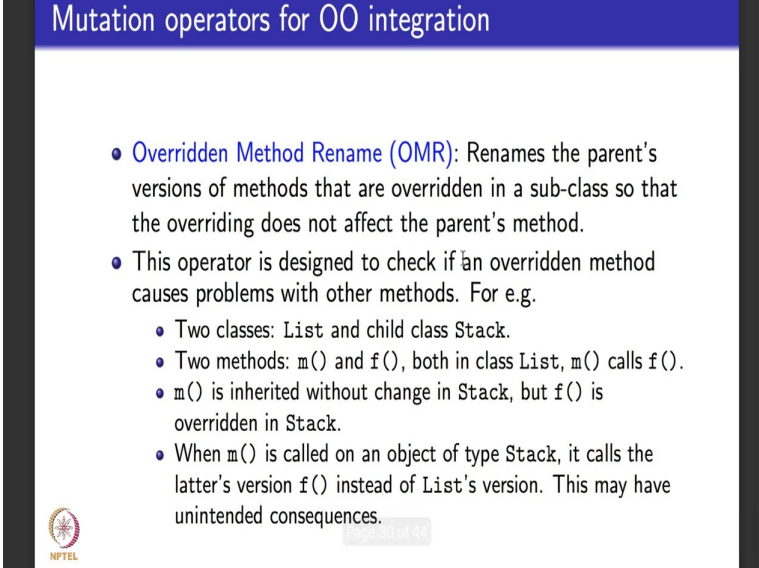
The next operators called overriding method moving abbreviated as OMM. What is it do? Each call to an overridden method is moved to the first and to the last statements of the method.

And wherever it is in the program, it is moved up by one statement, its moved down by one statement. The previous one we deleted the declaration of the method. Here were not

deleting it, instead there is a particular line in which the method is declared. What we do in the declaration of the call, we move its location within the program source code. We move it to the first statement, we move it to be the last statement, we move it one statement up, we move it one statement down.


What will this test? This will test the following. Typically overriding methods in child classes they often call the original method in the parent class for some reason. For example, they could call it to modify a variable that is private to the parent. There could be other reasons also. One common mistake is to make the call to the parents version at the wrong time. You called too early, maybe by the time you called it was too late to use the variable. So, moving around the call the statement of the call, we will be able to correctly detect whether it was called at the right time. So, killing such a mutant will detect errors related to where the call is made. If it is made at an inappropriate place it will be obviously detected. So, the next operator also deals with overridden methods. So, I will go back, we deleted overriding methods, we moved overriding methods.

(Refer Slide Time: 10:09)



Mutation operators for OO integration

- **Overridden Method Rename (OMR):** Renames the parent's versions of methods that are overridden in a sub-class so that the overriding does not affect the parent's method.
- This operator is designed to check if an overridden method causes problems with other methods. For e.g.
 - Two classes: List and child class Stack.
 - Two methods: `m()` and `f()`, both in class List, `m()` calls `f()`.
 - `m()` is inherited without change in Stack, but `f()` is overridden in Stack.
 - When `m()` is called on an object of type Stack, it calls the latter's version `f()` instead of List's version. This may have unintended consequences.

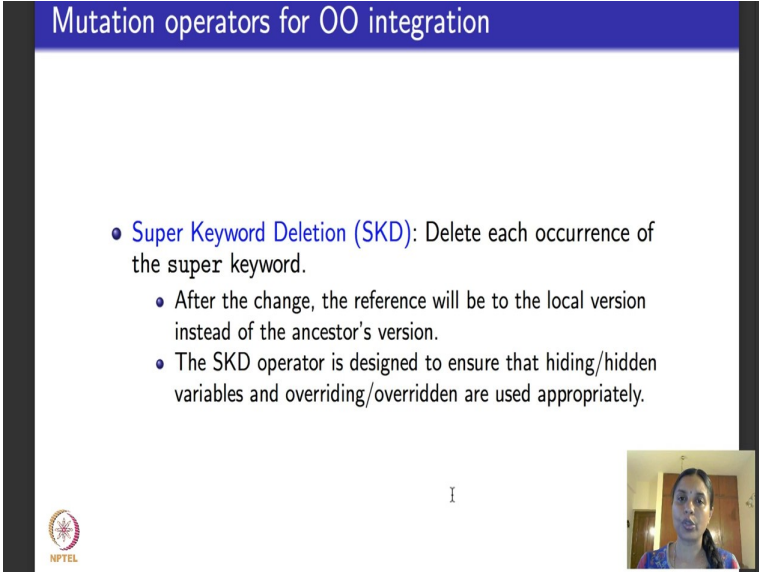
 NPTEL

Now, we are going to rename overriding methods, overridden method rename abbreviated as OMR. What does this do? It renames the parent's versions of the methods that are overridden in a subclass so that overriding does not affect the parent's method. What is this operator designed to check for? This mutation operator basically checks if an overridden method actually causes problems with other methods. So, here is an

illustrative example. Let us say there are two classes called list and stack. List is the parent class, list has a child class called stack. Maybe list is a list operations and stack is one of the ways of implementing a list, whatever it is. So, there are two classes: list, parent class and its child class called stack.

And there are two methods, a method called m a method called f. Both of them are present in the class list and in the class list method m calls the method f. In addition method m and f are also present in the class stack. How are they present in the class stack? m is inherited without a change in the class stack, but f is overridden in the class stack. So, now, when I rename one of them what will happen? When m is called on an object of let us say type stack, it calls the stack's version of f instead of the list's version. This kind of problems I can detect by using this mutation operator, I hope this is clear.

(Refer Slide Time: 11:40)



The slide is titled "Mutation operators for OO integration" in a blue header. It contains a bulleted list with the following items:

- **Super Keyword Deletion (SKD):** Delete each occurrence of the super keyword.
 - After the change, the reference will be to the local version instead of the ancestor's version.
 - The SKD operator is designed to ensure that hiding/hidden variables and overriding/overridden are used appropriately.

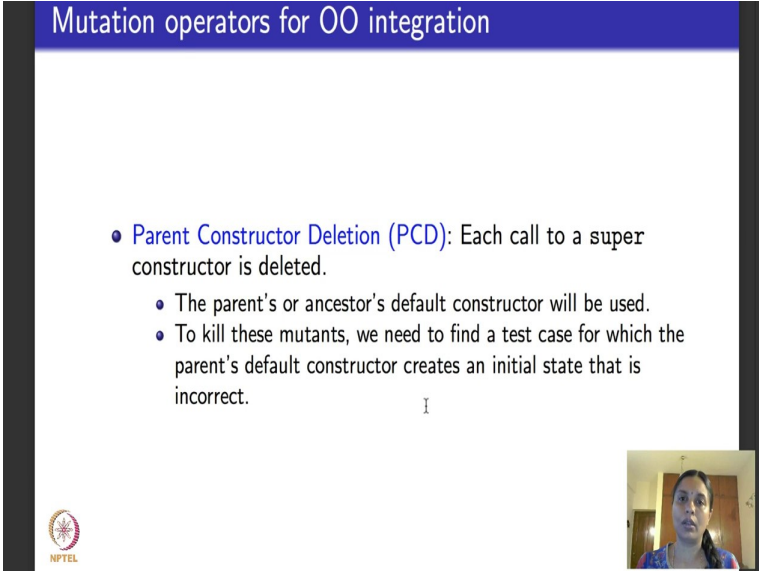
In the bottom right corner, there is a small video inset showing a woman speaking. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.

Moving on the next mutation operator we are going to see relates to this key word called super. If you remember we used super, why did we use it? We used it to refer to appropriate variables in an ancestor right, in an ancestor class. That is the context in which I had introduced this when I told you about object oriented features in the last lecture. So, super keyword deletion, what does it do? It just simply deletes the super keyword, which means what? After deletion the reference will be to the local version of the variable instead of the ancestor's version. So, without, so let us say if you had super

dot x, you delete the word super and directly refer to x. So, the x that I am referring to now is in the local version and not in the ancestor's version.

So, what kind of errors can this reveal? This can reveal errors that ensure that hiding or hidden variables or overriding or overridden methods or variables are used appropriately. So, if it was meant to be actually from the ancestors class, not the local version, then deleting it will use the local version and reveal a potential error.

(Refer Slide Time: 12:48)



The slide is titled "Mutation operators for OO integration" in a blue header. It contains a bulleted list with the following content:

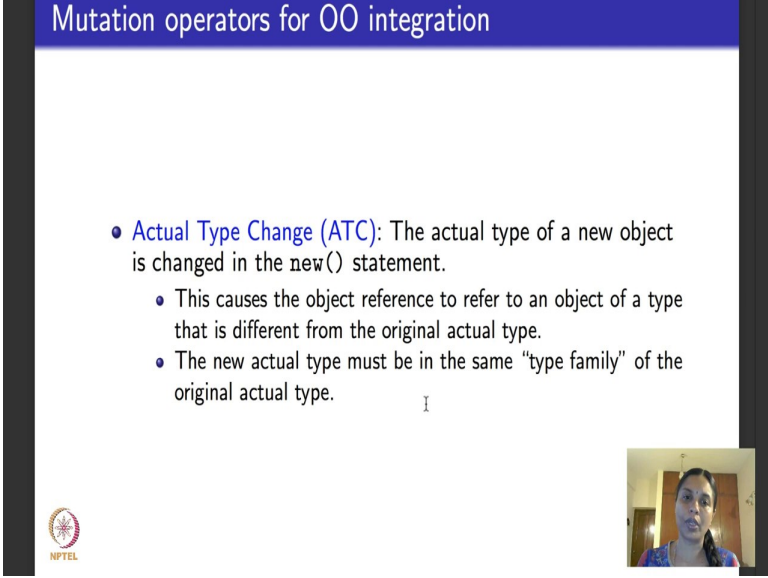
- **Parent Constructor Deletion (PCD):** Each call to a super constructor is deleted.
 - The parent's or ancestor's default constructor will be used.
 - To kill these mutants, we need to find a test case for which the parent's default constructor creates an initial state that is incorrect.

In the bottom right corner of the slide, there is a small video inset showing a woman speaking. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.

So, the next one next mutation operator is called parent constructor deletion. So, it deals with constructor. So, each call to the super constructor is deleted as the mutation operator describes. What happens in this case? In this case after deletion, the parents or the ancestors default constructor would be used.

Now to kill such mutants which delete the super constructor operator, what do we need to do? We need to find a test case for which the parents default constructor creates an initial state that is incorrect because that is when I will be able to observe an error about using the parents or an ancestors constructor. Otherwise I will not be able to observe any error.

(Refer Slide Time: 13:31)



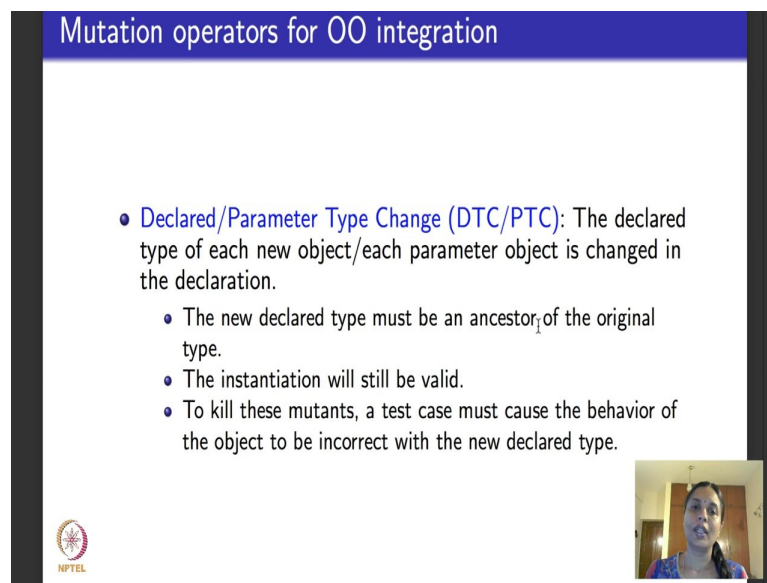
The slide has a blue header with the text "Mutation operators for OO integration". Below the header, there is a bulleted list. The first bullet point is "Actual Type Change (ATC): The actual type of a new object is changed in the new() statement." It has two sub-bullets: "This causes the object reference to refer to an object of a type that is different from the original actual type." and "The new actual type must be in the same 'type family' of the original actual type." In the bottom left corner, there is a small circular logo with the text "NPTEL" below it. In the bottom right corner, there is a small video inset showing a woman speaking.

- **Actual Type Change (ATC):** The actual type of a new object is changed in the `new()` statement.
 - This causes the object reference to refer to an object of a type that is different from the original actual type.
 - The new actual type must be in the same "type family" of the original actual type.

So, the next mutation operator changes types. Whose type that is changed, it changes the actual type of a new object, where does it change? It uses this new statement that is available in Java to be able to change the type of the new object.



What kind of problem this causes? This, if you do this then it will cause the object reference to refer to an object of type that is different from the original actual type. Is that clear, because I have changed it now in the new statement. So, now what will happen, the new actual type that it is acquiring must be the same, in the same type family of the original actual type,. I need to ensure this because you remember the good old thing about mutation. After mutating a program, the program must compile it should not be syntactically illegal. So, type change, if you do, then the type change better be consistent, so that the mutation can be applied through a ground string to get a new program that can still combine and then we can worry about whether we can strongly kill it or weakly kill it and so on.

(Refer Slide Time: 14:31)



Mutation operators for OO integration

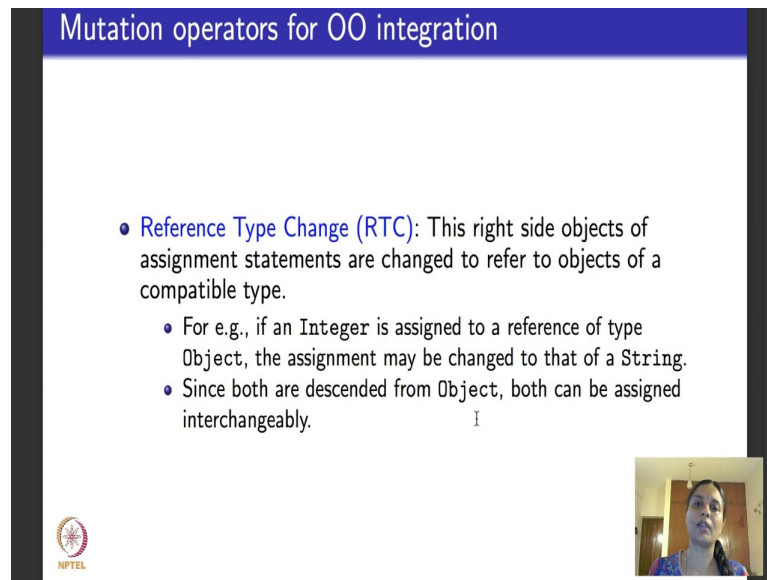
- **Declared/Parameter Type Change (DTC/PTC):** The declared type of each new object/each parameter object is changed in the declaration.
 - The new declared type must be an ancestor_T of the original type.
 - The instantiation will still be valid.
 - To kill these mutants, a test case must cause the behavior of the object to be incorrect with the new declared type.

So, the next operator in our long list is what is called declared or parameter type change. This one slide actually contains two mutation operators. Both the mutation operators are about changing a type. One changes the type of a declared type, one changes the type of a parameter. So, the declared type of each new object or each parameter object type is changed in the declaration. So, what will happen? The new declared type must be an ancestor of the original type and the instantiation will still be valid because that is how we do it.

To kill such mutants for where the declared type of a object or the parameter is changed, what should a test case do? It must cause the behaviour of an object to be incorrect with reference to the new change that I have made to the type. So, if that happens then I know that there is a problem with my original declaration or maybe my original declaration was fine. So, that is the integration feature that I test.

(Refer Slide Time: 15:28)



The slide is titled "Mutation operators for OO integration". It contains a bulleted list with the following content:

- **Reference Type Change (RTC):** This right side objects of assignment statements are changed to refer to objects of a compatible type.
 - For e.g., if an Integer is assigned to a reference of type Object, the assignment may be changed to that of a String.
 - Since both are descended from Object, both can be assigned interchangeably.

In the bottom right corner of the slide, there is a small video inset showing a woman speaking. The NPTEL logo is visible in the bottom left corner of the slide.

So, the next one that we are going to see is also a type change, but this is a type change called reference type change. What does this do? Here the reference is changed, which means the right side objects of assignment statements are changed to refer to objects of a compatible type. They reference right hand side of the assignment statement or the reference statement, the type of that is changed. Not the entire reference, but only its type.



For example, if an integer is assigned to a reference of type object then you could make a type change and change it to that of a string. In general it may not be allowed, integer changing to string there will be big problems related to compiling, but here it will work fine because both have descended from object, so both can be assigned interchangeably. And this will again detect any errors that relates to changing of types and the types of references.

Moving on, the next operator that we are going to deal with is related to overloading methods. In fact, we will work with overloading method change here, overloading method deletion here, two operators.

(Refer Slide Time: 16:24)

Mutation operators for OO integration

- **Overloading Method Change (OMC):** For each pair of methods that have the same name, the bodies are interchanged.
 - This ensures that overloaded methods are invoked appropriately.





So, what is overloading method change do? For each pair of methods that have the same name the bodies are interchanged, which means the code that corresponds to these methods, the line entire code that corresponds to these methods, they are swapped, they are interchanged. Then what is this will test, what sort of error this will test? This will test any error that relates to overloading methods being invoked properly If they are not invoked correctly at the same time then one is invoked instead of the other you will be able to kill the mutant and detect any errors related to proper invocation of overloaded methods.

(Refer Slide Time: 17:14)

Mutation operators for OO integration

- **Overloading Method Deletion (OMD):** Each overloaded method declaration is deleted, one at a time.
 - This operator ensures coverage of overloaded methods— all of them must be invoked at least once.
 - If the mutant still works correctly without the deleted method, there may be an error in invoking one of the overloading methods— the incorrect method may be invoked or an incorrect parameter type conversion has occurred.

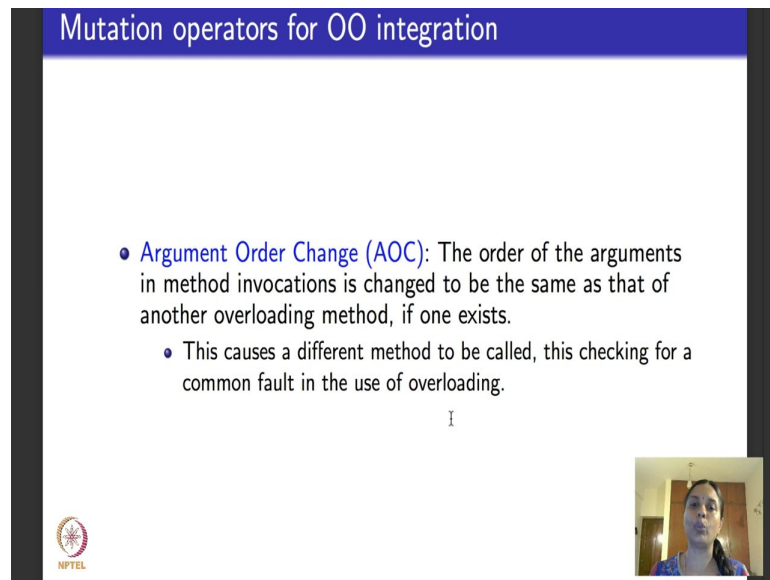


So, next operator also deals with overloading methods. Instead of changing them it deletes them. So, the operators called overloading method deletion abbreviated as OMD. What is this one do? Each overloaded method declaration is deleted. Please remember there is no point, suppose I have more than one overloaded methods in one shot I cannot apply this mutation operator and delete more than one. That will violate the problem of applying one mutation operator at a time to be able to mutate a ground string. This mutation operator can be applied to delete any overloaded method declaration but not more than one. If you apply it to more than one deletion then it will mean that you are deleting more than one at a time, that is not allowed. So, you delete each overloaded method declaration one at a time.

Choose which one to delete based on what you want to focus and test. What will this operator ensure? This operator will ensure coverage of overloaded methods. Why, because I delete then what will happen without it, then I test that. Then I put this one that I deleted back and delete something else then I test what happens without what second one that I deleted right. So, all the methods must be invoked once as and when they take turns to get deleted.

If a mutant still works correctly without the deleted method then it could be an error, it could be an error in invoking one of the overloading methods while doing integration. The incorrect method may be invoked or an incorrect parameter type conversion has happened. If nothing happens after deleting one particular method then maybe it was a piece of dead code and you could do without it right. So, if you get an equivalent mutant then it means that the method was dead code and you could do without it otherwise you will definitely be able to kill the mutant and observe one of these kind of errors.

(Refer Slide Time: 19:08)




Mutation operators for OO integration

- **Argument Order Change (AOC):** The order of the arguments in method invocations is changed to be the same as that of another overloading method, if one exists.
 - This causes a different method to be called, this checking for a common fault in the use of overloading.

I

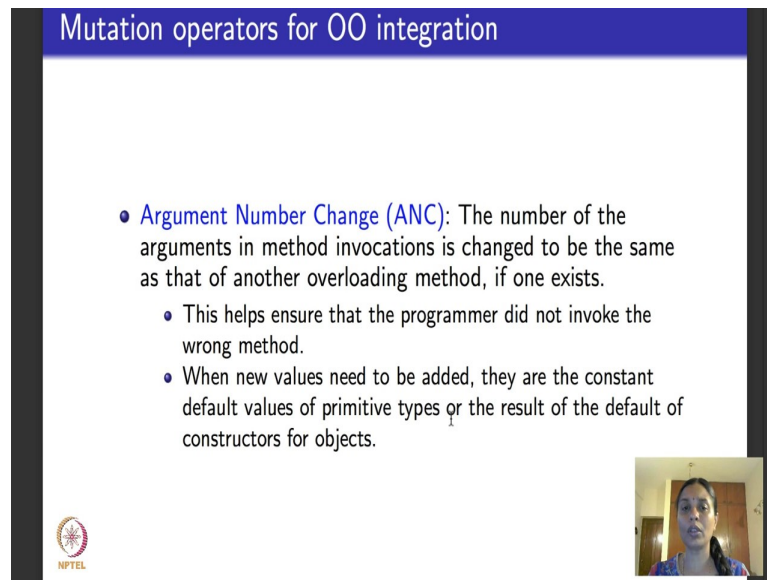
NPTEL



So, the next operator we will see changes the order in which arguments are passed during method invocation. It is called argument order change abbreviated as AOC, what does this do? The order of the arguments in a method invocation is changed to be the same as that of another overloading method if it exists. This causes a different method to be called because I am changing the arguments and hence it checks for common fault in the use of overloading. What happens? Because something could be called out of order, I have changed completely, so then, if there is an error because of the use of overloading I will be able to catch it.



So, the next mutation operator also deals with arguments.

(Refer Slide Time: 19:45)



Mutation operators for OO integration

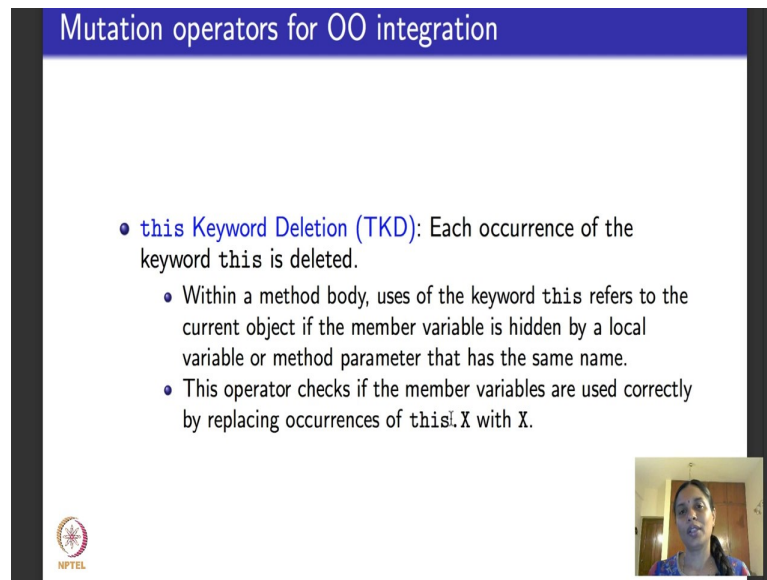
- **Argument Number Change (ANC):** The number of the arguments in method invocations is changed to be the same as that of another overloading method, if one exists.
 - This helps ensure that the programmer did not invoke the wrong method.
 - When new values need to be added, they are the constant default values of primitive types or the result of the default of constructors for objects.

But here we didn't change the order of arguments, here we change the number of arguments. Lets say if you had three, you could make that two, you could make that four that is what we mean change the number of arguments.

So, number of arguments in method invocations is change to be the same as that of another overloading method provided there is one such method that exists and you could replace it with that. What will this help? This will help to check that the programmer indeed invoked the correct method, he did not pick up the wrong method to invoke and when new values have to be added they are constant default values of primitive types or they are the result of default of constructors for object, is that clear.

(Refer Slide Time: 20:31)



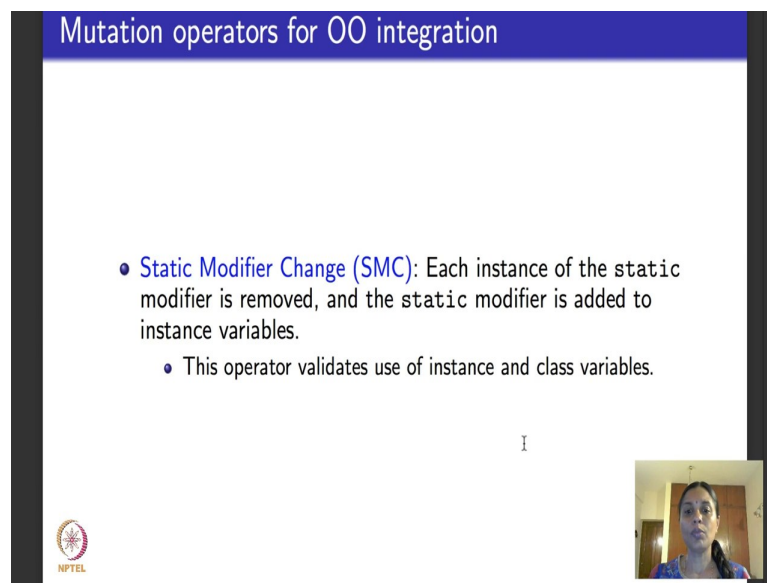
The slide has a blue header with the text "Mutation operators for OO integration". Below the header, there is a bulleted list. The first bullet point is "• **this Keyword Deletion (TKD):** Each occurrence of the keyword `this` is deleted." followed by two indented bullet points: "• Within a method body, uses of the keyword `this` refers to the current object if the member variable is hidden by a local variable or method parameter that has the same name." and "• This operator checks if the member variables are used correctly by replacing occurrences of `this.X` with `X`." In the bottom left corner, there is a small circular logo with the text "NPTEL" below it. In the bottom right corner, there is a small video inset showing a woman with dark hair, wearing a blue top, speaking.

- **this Keyword Deletion (TKD):** Each occurrence of the keyword `this` is deleted.
 - Within a method body, uses of the keyword `this` refers to the current object if the member variable is hidden by a local variable or method parameter that has the same name.
 - This operator checks if the member variables are used correctly by replacing occurrences of `this.X` with `X`.

So, moving on the next operator that we are going to see, we still have a long way to go remember I told you 20 operators. So, were probably around 15. So, we still have a good number of operators to go.

So, the next operator that we are going to see deletes this keyword called `this`. You remember Java has this keyword called `this`. What is the keyword `this` do? Within a method body, when I use a keyword `this` it refers to the current object of the member variable is hidden by a local variable or a method parameter that has the same name. Now if I delete it, then what will happen it checks basically, when I apply this mutation the resulting program if I am able to kill it then I am basically checking if the member variables are used correctly by replacing every occurrence of `this dot x` with `x`. So, here unlike the earlier one, when I say this keyword deletion, you have to be careful you have to delete more than one keyword of `this` because if you keep one and retain one they could be problems. So, sometimes you might have to delete one of this one, but if there is more than one related one, you have to delete all of them.


(Refer Slide Time: 21:38)



Mutation operators for OO integration

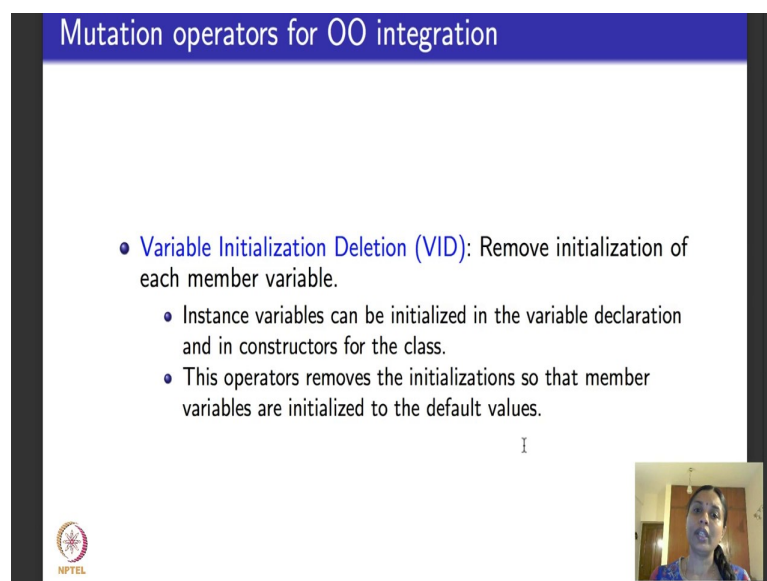
- **Static Modifier Change (SMC):** Each instance of the static modifier is removed, and the static modifier is added to instance variables.
 - This operator validates use of instance and class variables.

NPTEL



Then the next mutation operator is called static modifier change abbreviated as SMC. What happens here? Remember static variables, I told you right in the beginning instance variables and class variables of Java, class variables are static variables. So, each instance of a static modifier is removed and the static modifier is instead added to the instance variables. What will happen? This basically checks if the various instance variables and class variables that have been declared around a piece of code are declared correctly and are being used at the appropriate piece.


(Refer Slide Time: 22:13)



Mutation operators for OO integration

- **Variable Initialization Deletion (VID):** Remove initialization of each member variable.
 - Instance variables can be initialized in the variable declaration and in constructors for the class.
 - This operators removes the initializations so that member variables are initialized to the default values.

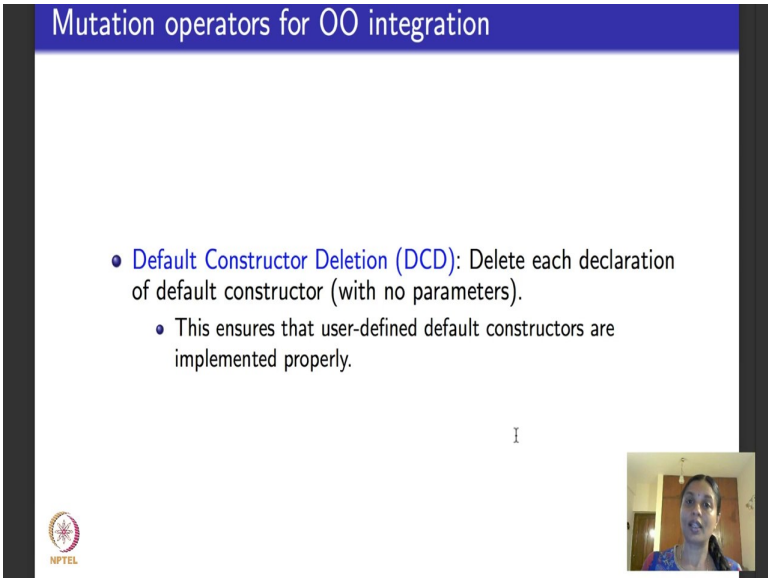
NPTEL



So, the next one variable initialization deletion VID, what does it do? It removes initializations of each member variable of a method or class. They are not initialized at all. If they are not initialized, what are the kind of errors that you could detect? Please remember instance variables can be initialized in the variable declaration or in the constructors for the class.

So, when I remove variables, this operator removes the initialization so that member variables are initialized to default values and it basically checks if the program will still run fine with reference to the initializations at the default values.

(Refer Slide Time: 22:49)

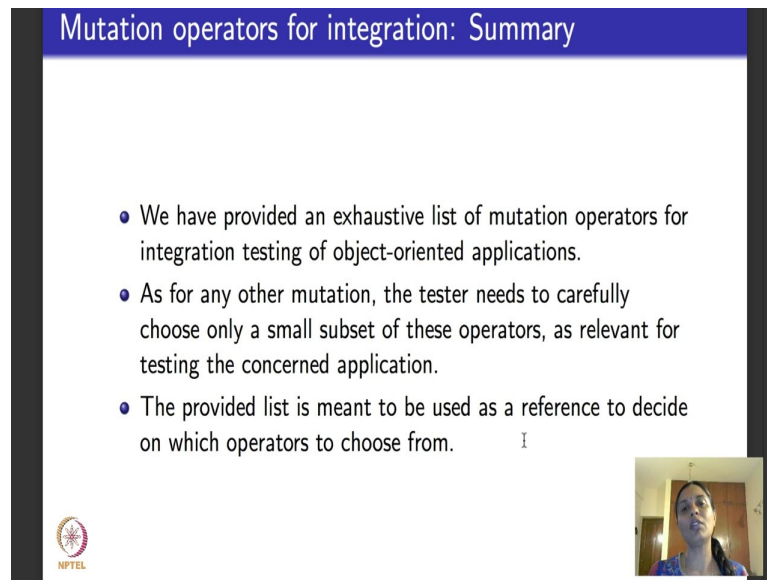


The slide is titled "Mutation operators for OO integration" in a blue header. The main content area is white and contains a bulleted list. The first bullet point is "Default Constructor Deletion (DCD): Delete each declaration of default constructor (with no parameters)." and the second bullet point is "This ensures that user-defined default constructors are implemented properly." In the bottom left corner, there is a small NPTEL logo. In the bottom right corner, there is a small video inset showing a woman speaking.

- **Default Constructor Deletion (DCD):** Delete each declaration of default constructor (with no parameters).
 - This ensures that user-defined default constructors are implemented properly.

So, the next, the last one, in our really long list of mutation operators what does it do, it is called a default constructor deletion DCD. It deletes each declaration of the default constructor with no parameters. What can this ensure? This ensures that the user defined default constructors are implemented properly. Suppose I delete and the delete, after deletion, the program still works fine which means it is an equivalent mutant then clearly I may not need it right. There is something wrong with my implementation. Suppose I delete an after deletion my program thus have a problem then I have implemented it correctly at least at the integration level.

(Refer Slide Time: 23:30)



The slide has a blue header with the title "Mutation operators for integration: Summary". Below the header, there are three bullet points in a list. In the bottom right corner of the slide, there is a small rectangular video inset showing a woman speaking. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.

- We have provided an exhaustive list of mutation operators for integration testing of object-oriented applications.
- As for any other mutation, the tester needs to carefully choose only a small subset of these operators, as relevant for testing the concerned application.
- The provided list is meant to be used as a reference to decide on which operators to choose from.

So, what did we do? We provided an exhaustive list. It must be tiring it must be a little confusing in fact, to go through such a big list. You might want to play this video once again to get the list right. But intuitively what it tells you, the list that we provided of twenty different mutation operators, it basically focuses on entities related to integrating methods, classes inheriting from each other, overloading, overriding, their access levels and it makes changes to all the places that focus on putting these together and tells you an exhaustive way of mutating to test this integration.

Obviously, when you have a piece of code that you want to test for design integration, you are not going to be able to consider all of these operators, not even half of them. Carefully based on the integration feature that you want to test, you will pick up one or two of these operators, maybe a few more and apply them to see the feature that you are going to test. So, the choice of the operator is based on the feature that you want to test and that is completely to be decided by you.

The focus of this lecture was to be able to provide an exhaustive listing to help you choose from this varied variety of choices that are available. So, I hope this exhaustive listing of mutation operators for object oriented integration testing helps you. You could use it when you are integrating java programs or C++ programs where all these features are available.

In the next lecture we will move on to using mutation testing for specifications, input grammars and XML models of inputs.

Thank you.



Software testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 43
Mutation Testing: Grammars and Inputs

Hello everyone, welcome to the next lecture of week 9. So, with this lecture I will finish mutation testing or syntax based testing.

(Refer Slide Time: 00:20)

Mutation testing for software artifacts: An overview				
	For programs	Integration	Specifications	Input space
BNF grammar	Programming languages Compilers	–	Algebraic specifications	Input languages like XML Input space testing
Summary				
Mutation	Programs	Programs	FSMs	Input languages like XML
Summary	Mutates programs	Tests integration	Model checking	Error checking





So, this is, this was what we had as an overview picture of mutation testing for various software artifacts. So I said through the span of my lectures, we would look at mutation testing along these lines.

(Refer Slide Time: 00:34)

Mutation testing for software artifacts: An overview				
	For programs	Integration	Specifications	Input space
BNF grammar	Programming languages	–	Algebraic specifications	Input languages like XML
Summary	Compilers			Input space testing
Mutation	Programs	Programs	FSMs	Input languages like XML
Summary	Mutates programs	Tests integration	Model checking	Error checking

Focus of this lecture: Mutation for input space grammars and XML.

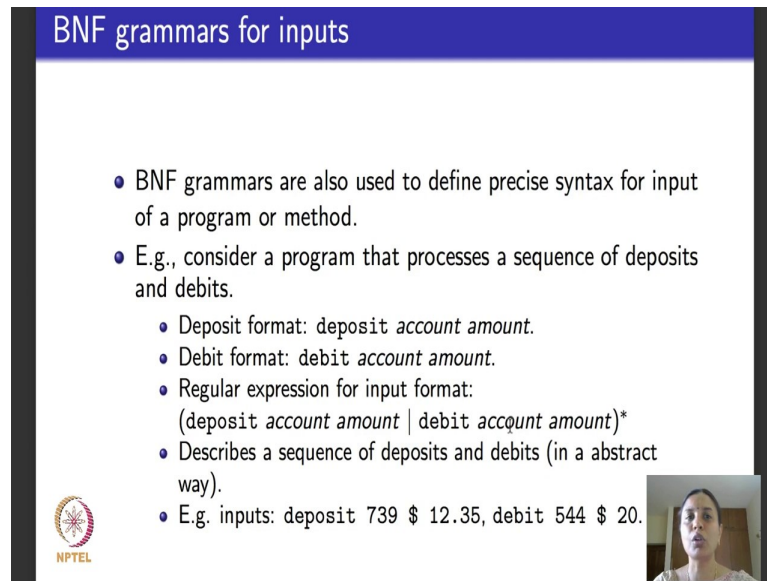


Here is an update of what we have done and where we are. So, what did we discuss? Mutation testing or syntax based testing is of two kinds: one directly based on grammar the other based on mutating programs. Grammar based testing can be applied for programs and mutation can be applied for programs. Grammar based testing cannot be applied for integration but mutation based testing can be applied for integration. Both grammar based and mutation based can be applied for specifications and again both grammar based and mutation based can be applied for directly testing input spaces.

What have we done? We completed all the things that are marked in green. We did grammar based testing for programming languages at an abstract level, I just told you what the coverage criteria on grammars are. We also did mutation testing for programs at the unit testing level and later on we specifically looked at mutation testing for design integration of generic kind and we went on and looked at mutation operators for specifically testing object oriented integration like, for a language like Java.

I told you that next we will do what is called input space partitioning based mutation with XML as one case study where I generate mutation based on grammars of input spaces. Specifically for grammars of XML formats for inputs as many languages. So, that will be the focus of this lecture. And as we continue to say the ones that I strike out we will not do because it is slightly outside the focus of this course.


(Refer Slide Time: 02:05)



BNF grammars for inputs

- BNF grammars are also used to define precise syntax for input of a program or method.
- E.g., consider a program that processes a sequence of deposits and debits.
 - Deposit format: *deposit account amount*.
 - Debit format: *debit account amount*.
 - Regular expression for input format:
(*deposit account amount* | *debit account amount*)*
 - Describes a sequence of deposits and debits (in a abstract way).
 - E.g. inputs: deposit 739 \$ 12.35, debit 544 \$ 20.

NPTEL



So, this lecture we are going to look at grammars for inputs. Grammars for inputs are again given in backus naur form as a context free grammar specifically or as a regular expression and when are they used, what are they used for, they used for defining precise syntax of how the inputs look like. Most of the times inputs may not be just a simple integer or a Boolean value or a floating point number. Inputs will have a lot of structure, input could be a entire webpage, the form that you filled into a web page the data that you read from a database, it could have a lot of structure. Inputs with a lot of structure are described using formal languages specifically regular expressions or grammars. For example, consider a program that operates as a part of a banking system and let us say that program processes the sequences of deposits and debits.

So, the format of deposit will look like this, it will say deposit which is like a keyword which tells you the action to do this amount into the account. Similarly the format for debit would be debit which is like a keyword, this amount into this account, is that clear. Now what are the transactions that you can do as a part of a bank account? You can either deposits and amount into an account or you can debit an amount into an account and you can repeat this transaction any number of times in any order. So, this is the regular expression that describes that input transaction sequence. So, it says the input transaction is given as a regular expression which is the star of two entities.

The first expression says deposit account amount which says deposit this amount into the account, the second expression says debit account amount which means debit this amount, the given amount, into the account. There is an 'or' connecting them which means you can do one of these actions and after that there is a star. Star in regular expressions as we saw indicates that you can do zero or more occurrences in any order of either this or this, which means you can either deposit a given amount into an account or you can debit a given amount from an account and any bank transaction is a sequence of inputs of arbitrary deposits and debits.

Now, this could be an input. How do you read this? You say deposit into the account number 739, this amount, 12.35 dollars. Similarly you could say debit into the account number 544, the amount 20 dollars.



(Refer Slide Time: 04:36)

BNF grammars for inputs, contd.

- Regular expressions might not suffice for some programs' inputs.
- We can use BNF grammars in that case to express a larger set of inputs.
- Grammar notation for the above example:


```

bank ::= action*
action ::= dep | deb
dep ::= 'deposit' account amount
deb ::= 'debit' account amount
account = digit3
amount ::= '$' digit+ '.' digit2
digit ::= 0|1|2|3|4|5|6|7|8|9
      
```

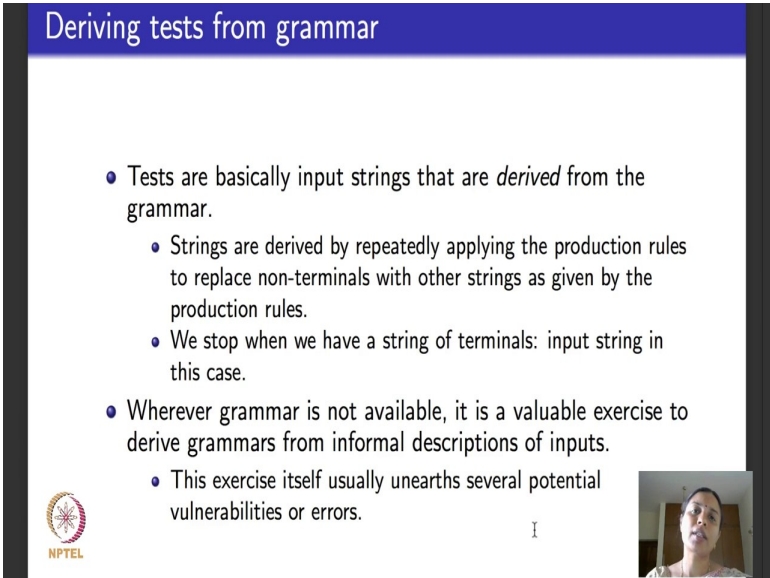



So, regular expressions, this is simply given as regular expressions, but sometimes as we saw regular expressions may not be enough and you might need slightly more expressive structures like context free grammars or arbitrary grammars which can be expressed in a normal form like Backus-Naur form. So, here is a same sequence of input transactions. In this slide I had given it to you as a regular expression. Here I am giving it to you as a grammar. How do I read it, I read it as follows.

So, what does a bank transaction? Bank transaction is a sequence of zero or more actions denoted by action star, then the next rule is action could be a deposit or a debit, dep for short form for deposit, deb short form for debit. A deposit is basically deposit the key word deposit some amount into an account number, debit is similarly the key word debit which depicts a fixed action of an amount into the account. Account is typically a 3 digit number, amount is given in dollars or any currency and its one or more digits followed by a decimal point followed by at least two digits, digits could be anything from 0 to 9 is that clear. So, I using this grammar I could derive an expression which looks like this.

So, I could say either deposit a particular amount which is this into this account what is it say, it says account is always a three digit number and amount is always in dollars. In this case, it has 0 it has one or more digits which is denoted by digit to the power of plus means 1 or more digits it cannot be 0 digits because you have to deposit or debit a nonzero number and it could also have decimal digits up to 2, decimal point two, is that clear.

(Refer Slide Time: 06:25)



The slide is titled "Deriving tests from grammar" in a blue header. It contains a bulleted list of points. In the bottom left corner is the NPTEL logo. In the bottom right corner is a small video inset showing a person speaking.

- Tests are basically input strings that are *derived* from the grammar.
 - Strings are derived by repeatedly applying the production rules to replace non-terminals with other strings as given by the production rules.
 - We stop when we have a string of terminals: input string in this case.
- Wherever grammar is not available, it is a valuable exercise to derive grammars from informal descriptions of inputs.
 - This exercise itself usually unearths several potential vulnerabilities or errors.

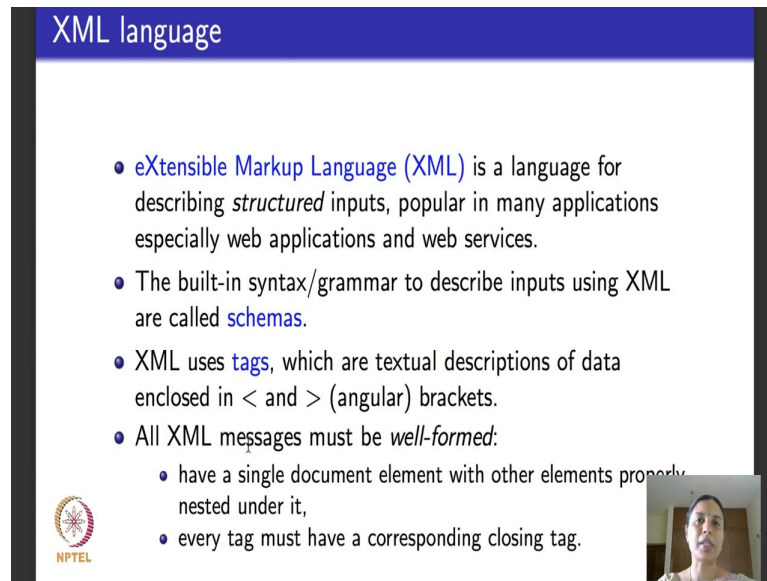
So, how do I derive tests from these grammars? Tests are basically input strings that are derived from the grammar. How do I derive? You follow the normal notion derivation that we saw for grammars when we began mutation testing last week.

So, what is the normal notion of derivation go like? You begin with the start symbol, designated start symbol and you use the production rules. Keep applying production rules by picking up one non terminal that is available at any given point in this string derived so far. Replace that non terminal by picking an appropriate rule that replaces that non terminal with another string. Keep doing this still you are left only with a string of terminals. In other words no more non terminals to replace. When you stop there that is the string that you derive and that belongs to the input space corresponding to the grammar that is given for that particular software artifact.

Whenever grammar is not available, you might wonder how are we going to use it? Usually it is very wise to be able to as a tester derive the grammar yourself from informal description of the input given. Some description or the other of the input will always be given because the software artifact and its input have to be described. Otherwise software development does not progress, but to be able to automatically derive tests you need grammar notations which are syntactically formal notations for describing inputs.

If it is not there then, as a tester, it is completely worth spending some time deriving the grammar from the informal description given. In the past empirical studies I have shown that just this exercise of deriving grammar from an informal description of the input sometimes unearths nontrivial bugs or vulnerabilities in the inputs to the software. So, it is a valuable exercise in its independent right and is also useful for mutating and producing inputs to a grammar.

(Refer Slide Time: 08:15)



The slide is titled "XML language" in a blue header. It contains a bulleted list of points about XML. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it. In the bottom right corner, there is a small video inset showing a person's face.

- eXtensible Markup Language (XML) is a language for describing *structured* inputs, popular in many applications especially web applications and web services.
- The built-in syntax/grammar to describe inputs using XML are called *schemas*.
- XML uses *tags*, which are textual descriptions of data enclosed in < and > (angular) brackets.
- All XML messages must be *well-formed*:
 - have a single document element with other elements properly nested under it,
 - every tag must have a corresponding closing tag.

We will see XML as a specific example because it is a very popular input format that is used across several different applications. So, what is XML? This module is not intended as an exhaustive introduction to XML please feel free to read up details about XML from other resources. I assume that you know some basics of it and I will use it mainly as a language to describe inputs and as a case study to show you how test cases are generated from the input.

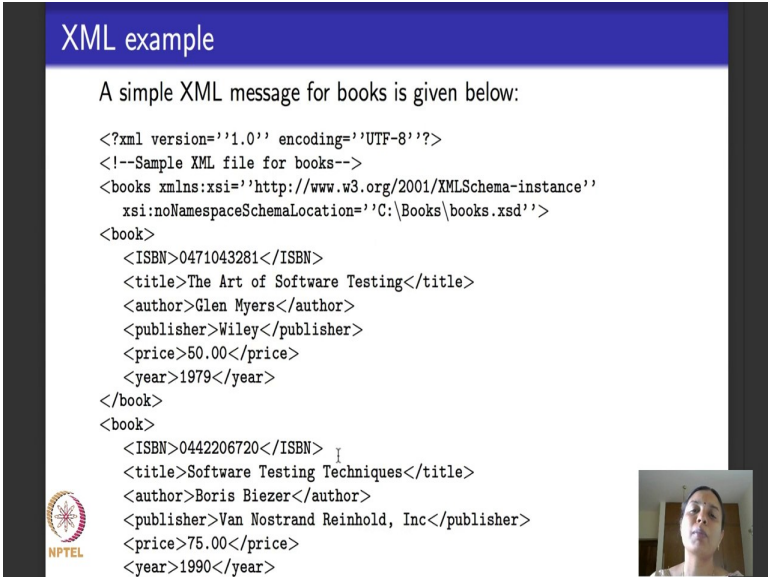
So, XML expands out to eXtensible Markup Language. It is a language for describing inputs that have a lot of structure in them, its popular across many applications typically its popular across web applications and web services. Of course, I basically work with a lot of embedded software design and development and if you see almost all the languages that I work with which is Simulink, AADL which are appropriately embedded design and architecture languages, they all have XML to describe their input language format. So, XML like all other structured formats, has a built-in syntax or a grammar to describe.

The built-in syntax or grammar that is available in XML it is what is called a collection of schemas. We will not introduce schemas thoroughly, but you can look up schemas from any standard book or reference on XML. In addition XML inputs use a notation called tags. What are tags? You can think of tags as some kind of textual description of data and the textual description of data is enclosed within angular brackets. Angular

brackets means this is the opening of the tag this is the closing of a tag. Each tag comes with this angular bracket and then for each open tag there is a corresponding closed tag. Inside each tag the some kind of a structured text that is described.

Typically we note that all XML messages must be well formed. What do we mean by that? They typically have a single document element with that other elements properly nested within it. There is a structure that is described in a nested fashion and every tag as I told you that has an opening tag, must also have a corresponding closing tag. So, in this sense if you know HTML which is an old language, it is a lot like HTML. It is another markup language, but of course, it has a more, a lot more sophisticated richer language to describe structured data formats.

(Refer Slide Time: 10:33)



XML example

A simple XML message for books is given below:

```
<?xml version='1.0' encoding='UTF-8'?>
<!--Sample XML file for books-->
<books xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
xsi:noNamespaceSchemaLocation='C:\Books\books.xsd'>
<book>
  <ISBN>0471043281</ISBN>
  <title>The Art of Software Testing</title>
  <author>Glen Myers</author>
  <publisher>Wiley</publisher>
  <price>50.00</price>
  <year>1979</year>
</book>
<book>
  <ISBN>0442206720</ISBN>
  <title>Software Testing Techniques</title>
  <author>Boris Biezer</author>
  <publisher>Van Nostrand Reinhold, Inc</publisher>
  <price>75.00</price>
  <year>1990</year>
</book>
</books>
```

NPTEL

So, here is a simple example of how an XML message for books will look like. So, what does it say? If you ignore the first four lines, we will come back to that it says you concentrate on this. So, it says book opening tag is here the corresponding closing tag is here. So, the opening tag is given by this angular brackets book, the closing tag is given by the angular brackets book with an additional forward slash here.

Now what are the a further structures that are available inside the tag book? Inside the tag book there could be several other tags ISBN number is one tag, title of the book is another tag, author is another tag, publisher is another tag, the price of the book is

another tag and the year in which the book was published is another tag. Of course, this is not exhaustive, you could have additional details describing the book like for example, you could say whether it is which edition, first edition, second edition, whether it is a paperback and so on.

And if you notice each of these sub tags have an opening tag and a closing tag and the syntax is fairly clear right. So, opening tag looks like this within angular brackets. After the text that comes inside the opening tag, there is a corresponding closing tag which is the same as opening tags, tag, has angular brackets in addition has a forward slash. So, it says each book tag has ISBN number, title, author, publisher, price and year. So, this describes one book, the book with this number, title art of software testing, author Glen Myers, publishers Wiley, price is 50 dollars, let us say year of publication is 1979.



Now there could be one more book tag which again gives the same details for another book. That book's ISBN number is this. the title of the book is called software testing techniques, author is Boris Biezer, publisher is Van Nostrand Reinhold, price is so much, year is 1990 and so on. So, and what is the top part, top part is some kind of a header information it tells you which is the version and the encoding of XML that you are using and this is a comment which tells you the this is a sample XML file meant to describe books and this is the place where I kept the schema from. In this schema books dot x s d, which I have kept here I can describe the following, which I have not given you the schema.

(Refer Slide Time: 12:47)

XML example

The following are described in the schema for the XML for books (actual schema not given):

- A books XML message can contain any number of book tags.
- Each book tag contains six pieces of information:
 - title, author and are strings.
 - price is of type decimal numeric, has two digits after decimal point, lowest value is 0.
 - ISBN has up to 10 numeric characters.
 - year is an integer with four digits.
- The criteria used for grammars can be used to derive XML messages for test inputs.



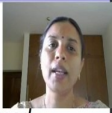
Please note that we have not described the schema, but I will tell you in English what the schema will describe in structured language of XML.

So, it says the schema will say the following it says this which begins here XML message can contain any number of book tags which are these. There is one book tag here, two book tags that I have shown. Each book tag contains 6 pieces of information title author and publisher, there are strings, price is a decimal numeric has two digits after the decimal point lowest value is 0. There will be a schema to describe this within XML I have not given you the schema.


Please remember that I have just described it in English ISBN is a number that has up to ten numeric characters year is an year with 4 digit and once I have the schema, the schema looks exactly like my grammar - it has terminals it has non terminals, it has rules that tell you how strings of non terminals and terminals can be derived. And so I can use the same criteria that I told you for grammars when we did that module to be able to derive XML messages for test inputs. In this case the test input will be a specific book or a specific set of books and I use the grammar to be able to derive those books. The grammar is obtained directly from the schema description of the XML message.

(Refer Slide Time: 14:10)

Mutation for input grammars



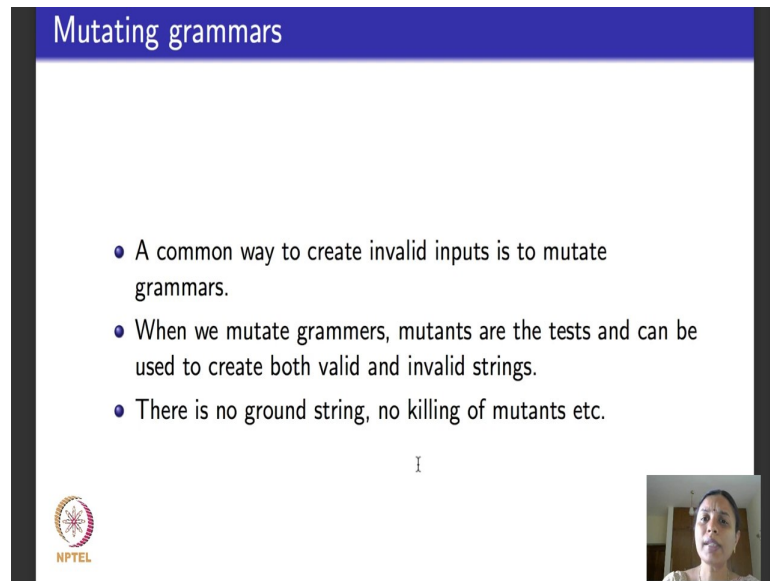
- While testing a program, tester needs to provide *malformed* inputs and see if the program *rejects* or *handles* them in a proper fashion.
- Such *invalid inputs* imply a lot about the functionality of the program and also check if the program/system handles faults properly.
- Security testing, safety critical system testing etc. all recommend or routinely test a system against invalid inputs.
- Input grammars can be mutated to generate invalid inputs as well.



Now, moving on when I consider mutation for input space grammars like XML or any other grammars while testing a program what is the tester, in addition to normal inputs the tester is this also useful if the tester provides malformed or invalid inputs. Why are invalid inputs necessary? Invalid inputs are necessary because it helps to check if the program does fault handling correctly. Invalid inputs also imply a lot about the core functionality of the program on correct inputs.

If the program works correctly on invalid inputs also then there is definitely wrong, something wrong with the functionality of a program. Typically in areas like security testing, testing for embedded systems, they insist that you do tests with invalid inputs at every level, specifically at system level and they have known to reveal a lot of vulnerabilities in the system. Just as we saw that standard mutation operators for input grammars can be used to generate valid inputs, mutation for input grammars can also be used to generate invalid inputs. Once you generate an invalid input you give it to a software artifact like a program and see how the program behaves on that invalid input.

(Refer Slide Time: 15:28)



The slide has a blue header with the title "Mutating grammars". Below the header, there are three bullet points:

- A common way to create invalid inputs is to mutate grammars.
- When we mutate grammars, mutants are the tests and can be used to create both valid and invalid strings.
- There is no ground string, no killing of mutants etc.

At the bottom left of the slide is the NPTEL logo, and at the bottom right is a small video inset showing a person speaking.



Now we will see what are the common ways to create invalid inputs by mutating grammars. Common ways to create valid inputs by mutating grammars, we already saw, when we introduced grammars. Now I will tell you how to create invalid inputs by mutating grammars. Please note that when we mutate grammars the mutants that we create are the test cases themselves. This was not the case when we mutated programs. When we mutated programs, we started with the program if you remember we called that program a ground string, we mutated that program, the resulting program that we got was another program that had to be tested.

We have to separately write test cases to kill the resulting mutant program. When we apply mutation for input space grammars the mutated string is directly the test case right. So, there is no notion of killing ground string, those things do not exist and the test case could be valid input, the test case could be an invalid input and that as is, acts like a test case.

(Refer Slide Time: 16:33)

Mutating grammars

- **Non-terminal replacement:** Every non-terminal symbol in a production is replaced by other non-terminal symbols.
- This is a generic mutation operator, several different invalid strings can be produced.
- In the example for deposit and debit, the rule `dep::='deposit' account amount` can be mutated to create
`dep::='deposit' amount amount and`
`dep::='deposit' account digit.`
- This gives the following tests: deposit 739 \$12.35 mutated to obtain deposit \$19.22 \$12.35 and deposit 739 1.



So, here are some standard operations to mutate grammar, what does grammar have? If you remember, grammar has non terminals, terminals, production rules and a designated non terminal called the start symbol. So, the mutation operators that we will see will basically work around these entities of a grammar.

So, the first mutation operator that we will see is what is called non terminal replacement. What does it do? Every non terminal symbol in a production rule in a grammar is replaced by other non terminal symbols. This is a very generic mutation operator. Remember typically large grammars for inputs typically XML grammars will have several different non terminals. So this says you can pick up any non terminal replace it with any arbitrary non terminal. So, it is like having probably dozens or even hundreds of rules just of this kind. As always while doing mutation you have to pick; which are the non terminals that I want to replace to create the invalid inputs that I want to test the program for.

And pick only those. Do not randomly apply non terminal replacement for replacing every non terminal, it does not make sense. For example, if you remember this grammar I will go back to that slide and show you the grammar for a minute. This was the grammar that we had which basically gave a sequence of input strings that either could debit an amount or deposit an amount from a particular account. So, for that grammar if

we had one of these rules which says a deposit is a string called deposit with the key word and it deposits an amount into the account, this, which are the non terminals here.



The non terminals could be amount or account. Deposit within quotes is like a terminal string, it is meant to be fixed its meant to indicate a keyword called deposit. So, the non terminals are amount an account. So, if I consider this mutation operator I can replace amount with account, amount with something else and so on that is what the first mutation I have done. The first mutation I have taken this production rule replaced this non terminal account with another occurrence of non terminal amount. So, instead of giving depositing an amount to an account you just give this invalid input which deposits an amount into an amount. So, we will see how the program that handles these input transactions will manage this input. Ideally it should have some exception handling to manage this another mutation.

That I could create would be to replace an amount with a single digit number, this would be a valid mutation, it will not produce an invalid string. Maybe it will be useful for testing what the program does on this particular valid input. So, these are just examples. Again non terminal replacement says that when you have a production rule in a grammar you could pick any non terminal and replace it with any non other non terminal both to produce valid inputs and invalid inputs. In this example we took this, it had two non terminals account and amount. First mutation that I created I replaced the non terminal account with amount, second mutation that I created I replaced this non terminal amount with the other non terminal that I had in the grammar which was digit. This resulted in an invalid mutant, this resulted in a valid mutant.

(Refer Slide Time: 19:44)

Mutating grammars

- **Terminal replacement:** Every terminal symbol in a production is replaced by other terminal symbols.
- This can again create several different mutants, some of them not appropriate.
- For the same e.g., the rule
`amount::='$', digit+ '.', digit2`
can be mutated to create
`amount::=',', digit+ '.', digit2`, `amount::='$', digit+ '$', digit2` and `amount::='$', digit+ '1', digit2`.
- This gives the following tests: deposit 739.12.35, deposit 739 \$12\$35 and deposit 739 \$12135.

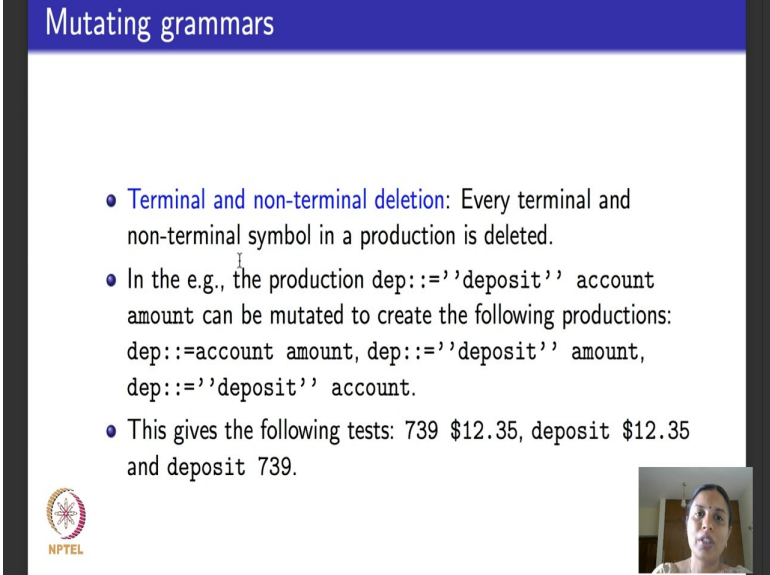


The next rule for mutating grammars would be instead of replacing non terminals you do terminal replacement, which means every terminal symbol in a production is replaced by other terminal symbols. Again this is a very generic rule, can create several different mutants, some of them completely junk, inappropriate, it is up to us as testers to pick and choose the one that we want wisely. For example, suppose I had this rule which was also a part of the grammar, amount equal to dollar and zero, one or more digit followed by a decimal point followed by exactly two digits, which basically gave the amount that could be deposited or debited with reference to an account. That can be mutated by replacing this dollar with a dot. That will create an invalid mutant, we will see what the program does.

Second kind of mutation replaces this dot with a dollar. We will see what happens because the ones within codes that terminals and I am considering terminal symbol replacement. So, I cannot replace digit, so I replace only the terminals. The third mutation, what is it do? It replaces the dot with a one. So, all these three mutations which replace one terminal symbol with some other terminal symbol create what are called invalid mutants, which could be used to test how the program that does, deals with these bank transaction handles these invalid mutants. So, if I apply the first mutant I will get a test case that looks like this. The program should figure out that this is junk input to handle it. Similarly if I apply the second mutant I get a test case which looks like this which is also junk input program should handle it. Third one is not a junk input it says

you deposit a fairly large amount into an account, if you have it why not the person is lucky.

(Refer Slide Time: 21:31)



The slide is titled "Mutating grammars" in a blue header. It contains three bullet points:

- **Terminal and non-terminal deletion:** Every terminal and non-terminal symbol in a production is deleted.
- In the e.g., the production `dep ::= 'deposit' account amount` can be mutated to create the following productions:
`dep ::= account amount`, `dep ::= 'deposit' amount`,
`dep ::= 'deposit' account`.
- This gives the following tests: 739 \$12.35, deposit \$12.35 and deposit 739.

At the bottom left is the NPTEL logo, and at the bottom right is a small video inset showing a person speaking.



So, the third kind of production rules says you delete the terminals and the non terminals. Every terminal and every non terminal in a production is deleted. Of course if you delete the whole thing you left no grammar left. So, you pick and choose a subset that you would want to delete and see when the restricted grammar is applied it will generate obviously, lesser input strings, what happens to that. For example, there was this production rule the that we had in the grammar for bank transactions. This can be mutated to create the following: I just delete the terminal deposit from that rule, that is the first mutant that I get. Second mutant, I delete the account from the rule, so I just get deposit amount. Third mutant, I delete the amount from the rule, so I get deposit account.

All three of them will produce junk or invalid or malformed test cases. We have to figure out what the program does. If the program does not handle exceptions on these test cases we have to build in those exception handlers.

(Refer Slide Time: 22:27)

Mutating grammars

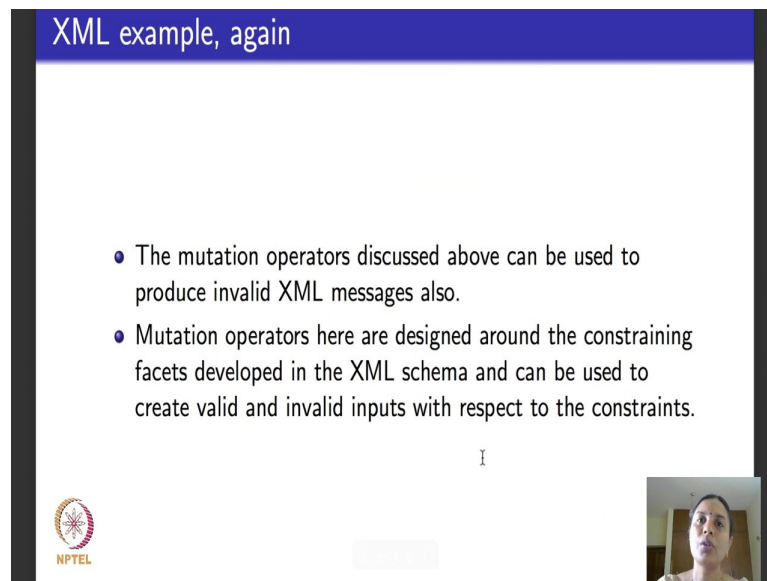
- **Terminal and non-terminal duplication:** Every terminal and non-terminal symbol in a production is duplicated.
- This mutation is sometimes called *stutter* operator.
- In the e.g., the production $\text{dep} ::= \text{'deposit' account amount}$ can be mutated to create the following productions:
 $\text{dep} ::= \text{'deposit' 'deposit' account amount},$
 $\text{dep} ::= \text{'deposit' account account amount}$ and
 $\text{dep} ::= \text{'deposit' account amount amount}.$
- This gives the following tests: deposit deposit 739 \$12.35, deposit 739 739 \$12.35 and deposit 739 \$12.35 \$12.35.



The next kind of production rule is, instead of deleting them you duplicate terminals and non terminals. Duplicate means what, you create another copy of one terminal or one non terminal. Again please remember always in mutation testing only one mutation at a time. So, do not duplicate two terminal symbols or in this in this case, do not delete two non terminal symbols or terminal symbols, only one operator at a time. So, because I duplicate, the term for duplication in computer science theories called stuttering.

So, the duplication operator is also called a stutter operator. For example, if I consider the same thing deposit which is deposit and into an account and amount, it can be mutated by replicating the word deposit twice, by replicating the non terminal account twice, by replicating the non terminal amount twice. All the three cases, except for maybe the, no sorry not the last one, the first two cases it will create invalid test cases because if the program cannot handle two word deposits, two keyword deposits. Second case the program cannot handle two account numbers. In the third case the program may or may not handle the amount based on whether there is a space or not. If you include this space, the program will consider it as an invalid input. All the three cases you all get invalid input except maybe for the third case where sometimes you all get valid input and this will again test if the program is handling invalid inputs correctly. It should not do anything, it should raise an exception and stop.

(Refer Slide Time: 23:56)



The slide has a blue header with the text "XML example, again". Below the header, there are two bullet points:

- The mutation operators discussed above can be used to produce invalid XML messages also.
- Mutation operators here are designed around the constraining facets developed in the XML schema and can be used to create valid and invalid inputs with respect to the constraints.

At the bottom left of the slide is the NPTEL logo. At the bottom right is a small video inset showing a person speaking.



So, now all these rules that we saw, four rules which is terminal replacement, non terminal replacement, terminal and non terminal duplication, terminal and non terminal deletion, they can be applied for XML example that we saw again. For the sake of succinctness, I have not really worked out an XML example. Feel free to ping me in the forum if you have any doubt and I will be able to explain it to you.

When specifically for XML people usually design mutation operators that check for constraints, like in the earlier thing we saw right that ISBN must be a 10 digits numeric string. So, they will make ISBN 0 empty string and see how the program handles it and so on. So, they are designed to work around the constraining artifacts to create invalid inputs with reference to the constraints.

(Refer Slide Time: 24:44)

Mutation testing: Summary

- Mutation or syntax-based testing is a powerful testing technique.
- Mutation operators are based on the underlying grammar of the software artifact.
 - Grammar always available for programming languages.
 - Might have to be derived for inputs.
- Mutation applied for programs need to be killed by test inputs.
- Mutation applied for inputs are tests themselves.
- Check the tool muJava available at: ^I
<http://cs.gmu.edu/~offutt/mujava/> to do mutation for Java programs.



So, here is a summary of mutation testing. This will be the end of mutation testing that we will see. Mutation or syntax based testing is a very powerful testing technique as I told you, it subsumes several other coverage criteria as we saw in one lecture module. They are based on the underlying grammar of the software artifact. For every programming language, grammar is available. For inputs whenever grammar is not available, it is a worth exercise to derive the grammars for these inputs.

Mutation when applied to programs result in mutated programs, they have to be killed by designing test cases. We saw two notions of killing: strong and weak killing whereas, mutation when applied to inputs directly result in test cases. So there is no notion of ground strings or killing mutated programs. And specifically if you are interested in doing mutation for Java, I recommend that you check out this tool that is available as a part of a project from George Mason and a University in China, it is a pretty good tool to do mutation for Java programs.

I also know of a couple of other tools for mutation for XML and for old Fortran programs which is a very good tool called Mothra, but feel free to Google for mutation tools for other kinds of languages. So, in the next lecture I will give you a summary of what we have done till now.

Thank you.

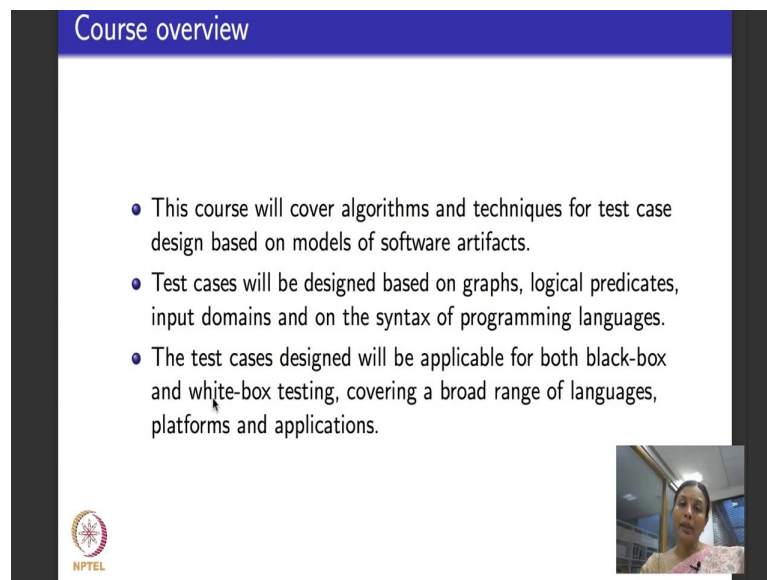
Software testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 44
Software Testing Course: Summary after Week 9

Hello there, we are in the last lecture of week 9. What have we been doing for the past two weeks? If you remember I have been doing syntax based testing or mutation based testing for the past two weeks. This sort of pretty much brings us to the end of the core of test case design algorithms that I wanted to teach you as a part of the course. At this point we are done with three-fourth of the course, we have three more weeks of lectures pending and it is a very good time to spend one video trying to look at what we learned from the beginning and where are we going to go from now.

So, that is what this lecture is going to capture, it is going to give you a summary of the course as it is after week 9.

(Refer Slide Time: 00:52)



The slide is titled "Course overview" in a blue header. It contains a bulleted list of three points: "This course will cover algorithms and techniques for test case design based on models of software artifacts.", "Test cases will be designed based on graphs, logical predicates, input domains and on the syntax of programming languages.", and "The test cases designed will be applicable for both black-box and white-box testing, covering a broad range of languages, platforms and applications." In the bottom left corner is the NPTEL logo, and in the bottom right corner is a small video inset showing Prof. Meenakshi D'Souza.

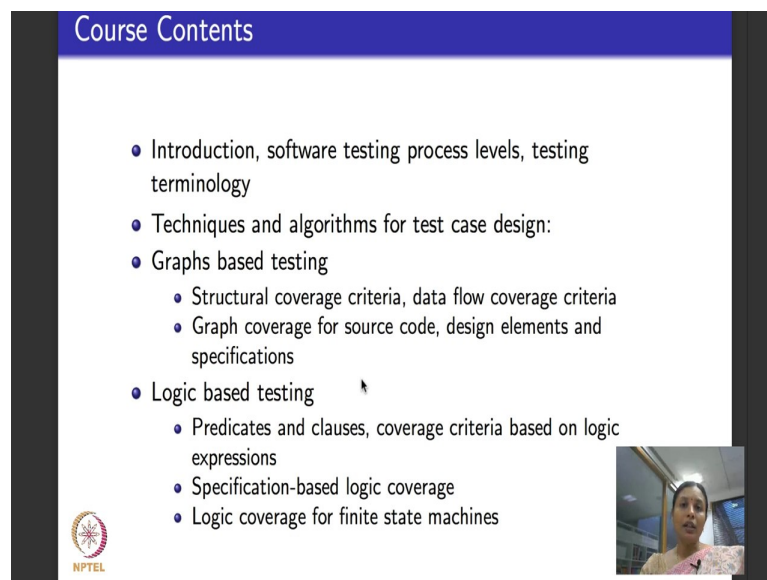
- This course will cover algorithms and techniques for test case design based on models of software artifacts.
- Test cases will be designed based on graphs, logical predicates, input domains and on the syntax of programming languages.
- The test cases designed will be applicable for both black-box and white-box testing, covering a broad range of languages, platforms and applications.

So, in the beginning, when I started lecturing at the beginning of the semester, what did I commit to. I told you, this is exact replica of the slides that we had shared at that time, I told you that will cover algorithms and techniques for test case design. How are we going to do it? We going to take software artifact create models or structures out of them and based on these models or structures we are going to see techniques for test case

design. And then what are the structures that we see: we saw graphs as models of artifacts, we saw logical expressions, then we only focused on the input space. And then finally, we focused on the syntax or grammar of the software artifacts.

The test cases that we designed can be used for black box when it is based on requirements and can be used for white box when it is based on source code.

(Refer Slide Time: 01:40)



Course Contents

- Introduction, software testing process levels, testing terminology
- Techniques and algorithms for test case design:
- Graphs based testing
 - Structural coverage criteria, data flow coverage criteria
 - Graph coverage for source code, design elements and specifications
- Logic based testing
 - Predicates and clauses, coverage criteria based on logic expressions
 - Specification-based logic coverage
 - Logic coverage for finite state machines

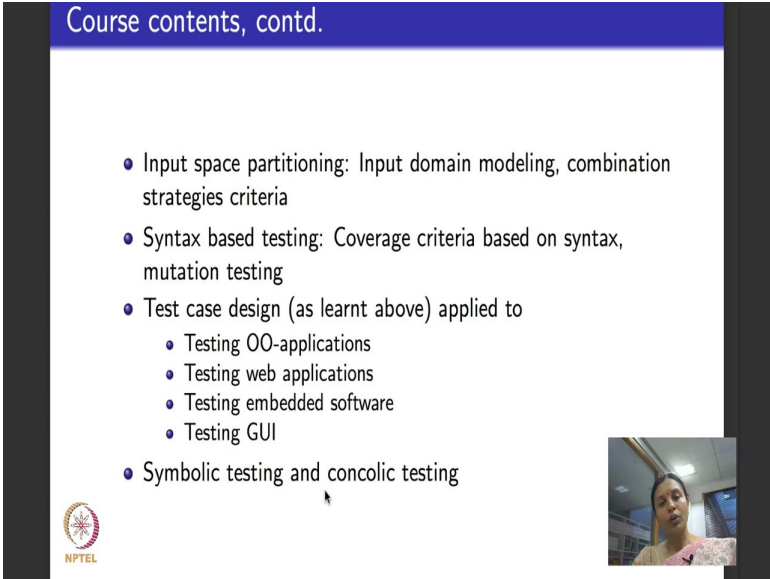
NPTEL

And then these were the contents of the course that I presented to you at the beginning of the lecture series right in the first week. So, we did begin by introducing what software testing is, motivating it, then I told you what are the process levels, what are the various set of jargons or terminologies that you will encounter in testing. After doing that initial bit, the main part of the course the technical part of the course that we focused on was related to techniques and algorithms for test case design.

We did graph based testing. We did two kinds of coverage criteria purely based on graphs - structural coverage criteria, data flow criteria. Along with this I gave you a good amount of basic graph traversal algorithms, depth first search, breadth first search, strongly connected components, topological sort and so on. And then what we did, we took graphs for source code told you how to draw control flow graphs for various design entities. We took graphs for design elements which were basically call graphs, sequencing constraints and so on. We took graphs for specifications and then saw how to apply these coverage criteria on these graphs.

Then we moved on to a next structural model, logic based testing. I introduced you to the basics of logic: predicates, clauses and then we looked at coverage criteria based on logic. We saw the most useful coverage criteria there are predicate coverage, clause coverage and active clause coverage. Then we applied logic based coverage criteria to source code for specifications and on finite state machines.

(Refer Slide Time: 03:11)



Course contents, contd.

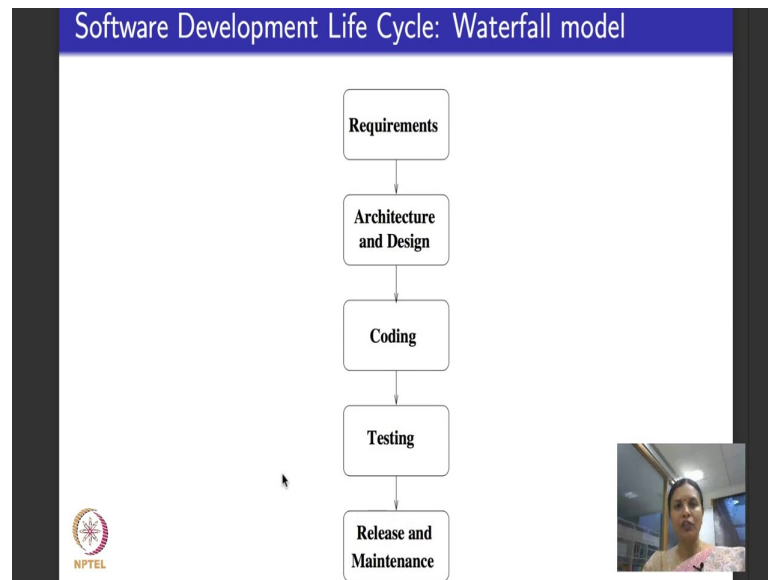
- Input space partitioning: Input domain modeling, combination strategies criteria
- Syntax based testing: Coverage criteria based on syntax, mutation testing
- Test case design (as learnt above) applied to
 - Testing OO-applications
 - Testing web applications
 - Testing embedded software
 - Testing GUI
- Symbolic testing and concolic testing

NPTEL

We moved on we looked at black box testing so to say, there we focused only on the inputs and outputs, the space of inputs and then we modeled the inputs and wrote test cases for coverage criteria based on various characteristics of the input domain. Then in the past two weeks I have been focusing on syntax based testing which focuses on the underlying grammar of the programming language or the software artifact, mutates or makes changes to that and writes test cases to kill the mutants.

Then the course in the beginning, as I committed to you I said, you will learn how to test object oriented applications how to test web applications, embedded software, GUI graphical user interface with or without using these criteria. We will see the popular state of the art issue mainly like a survey of each of these categories. Finally, I would like to end the course by presenting symbolic and concolic testing to you.

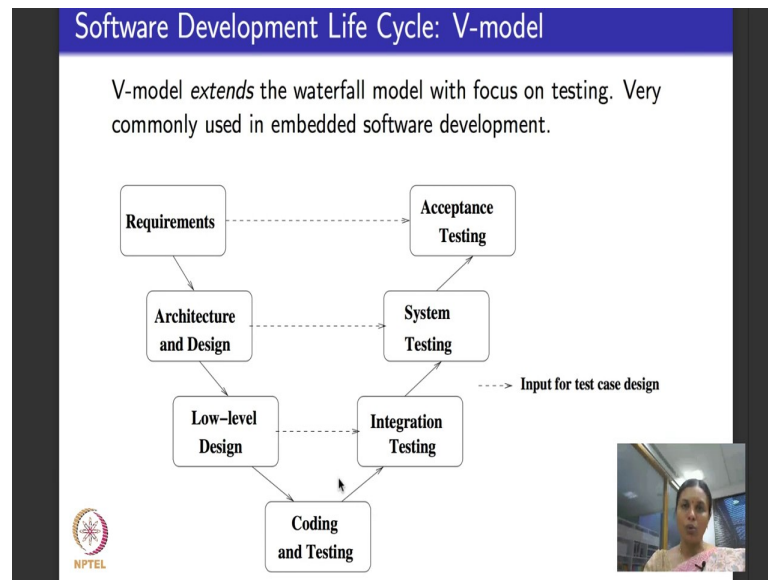
(Refer Slide Time: 04:09)



So, these were the course contents exactly as I had given you in the beginning of the course. Now I would like to spend some time looking back and thinking about what we have done, how does it relate to software development. Here is a popular model of software development that is still predominantly used even in this agile development world. It is called the waterfall model good old software development lifecycle model. Begins with requirements, goes on to do architecture and design, then people write code, they also unit test their code then they put the things together and do integration testing, system testing, collectively called as testing phase. Waterfall does not really distinguish between the kind of testing that you do. Finally, after the software is ready, its released and then maintained.

In the waterfall model you assume that requirements are baselined before you do architecture and design, you finalize the architecture and design before you write coding and when you have begun testing your code is more or less ready. There is not much going back, up and down, back and forth, that does not happen.

(Refer Slide Time: 05:11)



Testing as we saw in this course is predominantly related to another software development lifecycle model called V-model which is what is given in this cycle, in this slide. So, what is V-model do? It takes the waterfall model as it exists here and sort of bends it to look at the other side. It focuses into testing, splits the testing to see what are the various stages of testing that will go on? Does not really particularly focus about release and maintenance its of course, there, but the model that depicts the level development lifecycle let us go this part release and maintenance. So, takes requirements architecture and design, coding, takes testing and elaborates testing to let it span across other phases. This is how it looks like. So, it always begins with requirements like in waterfall model followed by architecture and design, again like in waterfall model, followed by I put this extra term where we model tries to focus on called low level design.

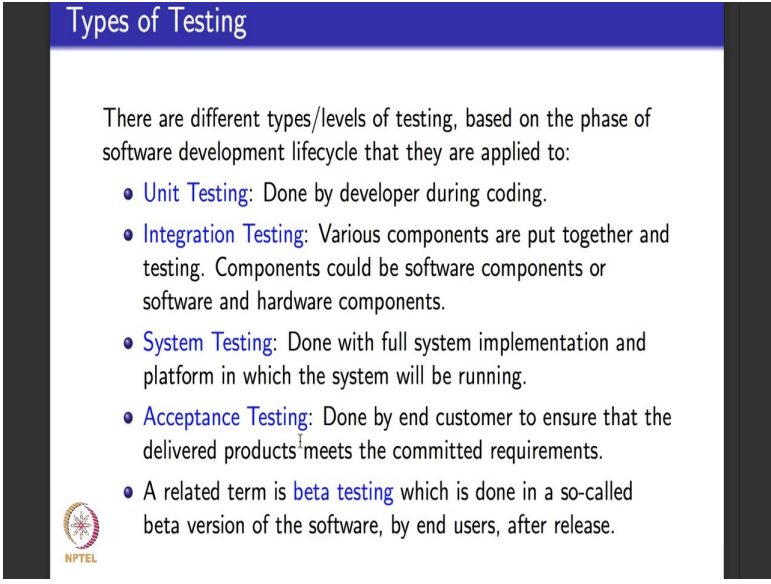
So, here we do system level design, here we design into the system components, each individual things is almost like code. Then people do coding, along with coding they do unit testing of the code. After that they do integration testing where they put together the software components and test it, and after they put together and tested the software components they put the hardware along with the software and then do integration testing and then they put the whole thing in this system and do system testing. Finally, when the system is ready, they do what is called acceptance testing. We have seen all

these terms before. What is the V-model say? Let us look at these dashed lines with arrows in the V-model.

It says that integration testing is related to low level design, system testing is related to architecture and design, acceptance testing is related to requirements. What is the legend for this dashed arrow? It says input for test case design that is if you want to design test cases for integration testing where you get data or inputs to design your test cases from? You get it from the low level design. Similarly if you want to design test cases for system level testing, the source set of software artifacts that give you inputs to design your test cases come from architecture and design documents. Similarly for acceptance testing which is mainly black box testing the source design for test cases come from your requirements doc.

What did we focus on in this course? We focused on the V-model. In particular we focused on how to use these various inputs from various software artifacts across the development lifecycle to be able to design test cases that we could use for coding and unit testing, we could use for integration testing, system testing or acceptance testing.


(Refer Slide Time: 07:58)



Types of Testing

There are different types/levels of testing, based on the phase of software development lifecycle that they are applied to:

- **Unit Testing:** Done by developer during coding.
- **Integration Testing:** Various components are put together and testing. Components could be software components or software and hardware components.
- **System Testing:** Done with full system implementation and platform in which the system will be running.
- **Acceptance Testing:** Done by end customer to ensure that the delivered products meets the committed requirements.
- A related term is **beta testing** which is done in a so-called beta version of the software, by end users, after release.

 NPTEL

This is again what I told you right in the first week, this is just a recap. These were the various types of testing we discussed unit testing and integration testing just now and after that come system testing and acceptance testing which we are all discussed as a part of the V-model. We also discussed what is called beta testing.


(Refer Slide Time: 08:15)

Testing Methods

There are two broad methods of testing:

- **Black-box testing:** A method of testing that examines the functionalities of software/system without looking into its internal design or code.
- **White-box testing:** A method of testing that tests the internal structure of the design or code of a software.

Black-box testing	White-box testing
Unit, integration, system and acceptance testing	Unit, integration and system testing
Usability, performance, beta and stress testing	



And I also told you at the beginning of the course that good old testing terminology says that there are two broad methods of testing: black box and white box. Black box does not look into the structure of the design or the code, assumes the design or the code to be a black box, designs test case is purely based on inputs and requirements. White box designs test cases by exploiting the structure of the design or the code. So, black box testing applies for unit testing, integration testing system and acceptance testing. White box testing also applies at unit testing, integration testing and systems testing phases. In addition a whole set of other testing usability, performance, load, beta testing, stress testing, they are all black box testing techniques.

Back in the V-model they happen here, at system testing or acceptance testing. Test cases for these kind of entities which test for non functional qualities of a software come from architecture and requirements and design. We did not look at any of those in this course. I will teach you, tell you a bit of GUI but not the rest.

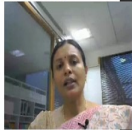

(Refer Slide Time: 09:20)

Types of test activities

Testing can be broken into four activities:

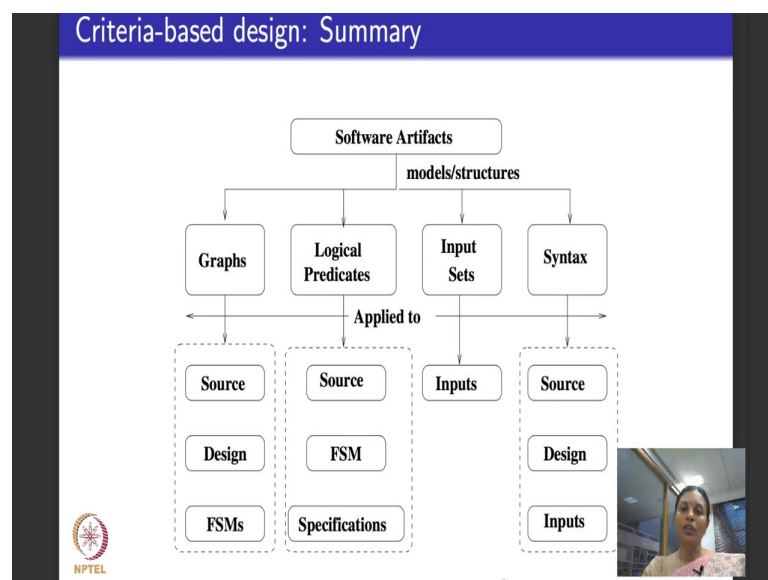
- Test design: Criteria-based on graphs, logical predicates, inputs and grammar.
- Test automation: JUnit.
- Test execution
- Test evaluation

Adjacent activities include test management, test maintenance and test documentation.



We also discussed types of test activities, just begin with test case design followed by automation, execution and evaluation. What did we do in the course as far as types of activities are concerned? We focused a lot on test case design basically criteria based test case design and I told you how to do automation in execution using the tool called JUnit which applies for Java programs.

(Refer Slide Time: 09:44)



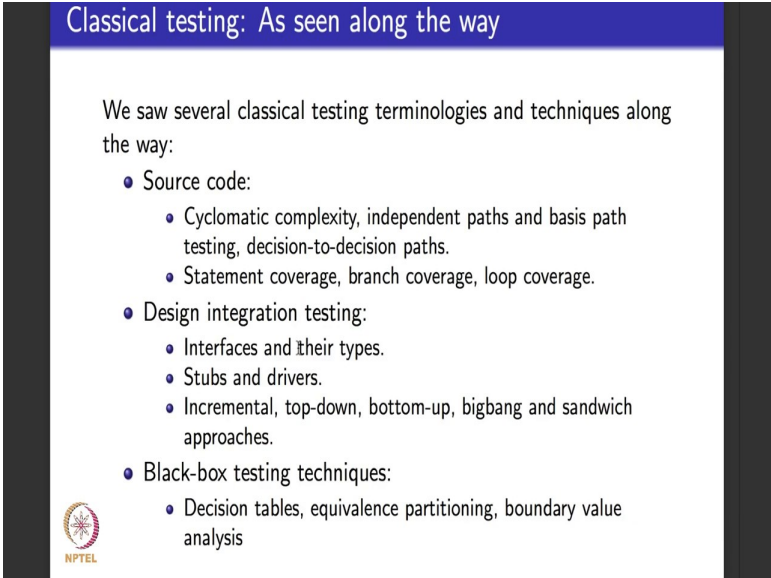
This slide gives a very good summary of what we have done exactly till now in the course. So, we took software artifacts which could be source code integrated source code

which is large code that includes all the modules put together, methods put together, classes put together. We took design elements, we took requirements either as documents or as finite state machines, then we took inputs, details about inputs. Then we chose four different kinds of model or structures. We modeled these software artifacts as graphs, defined coverage criteria over graphs. Two kinds structural and data, and then so how to apply it to source code, to design elements and to finite state machines

After that we took logical predicates which come from decision statements in the source code and from requirements or guards in finite state machines and define coverage criteria on logical predicates clause, predicate, combinatorial coverage, active and inactive coverage criteria. We applied it again to test source code, finite state machine and specifications. Then we focused on black box testing only on input spaces, define criteria that partition the input spaces and apply it to test the software artifact based on the inputs.

Finally, we did mutation testing which was syntax based testing. We saw how to apply to source code both within a method and at the software integration level. We saw how to apply it to design integration specifically we saw one module on object oriented design integration, finally, we saw how to apply to grammars for inputs.


(Refer Slide Time: 11:24)



Classical testing: As seen along the way

We saw several classical testing terminologies and techniques along the way:

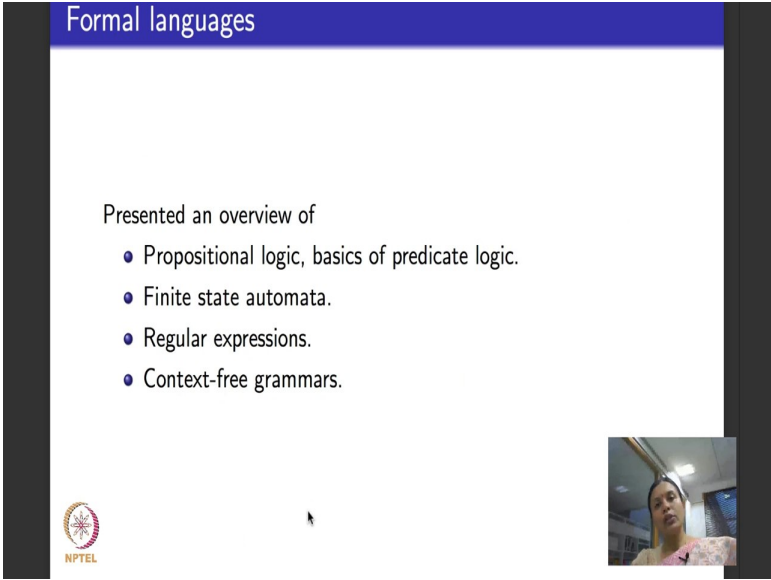
- Source code:
 - Cyclomatic complexity, independent paths and basis path testing, decision-to-decision paths.
 - Statement coverage, branch coverage, loop coverage.
- Design integration testing:
 - Interfaces and their types.
 - Stubs and drivers.
 - Incremental, top-down, bottom-up, bigbang and sandwich approaches.
- Black-box testing techniques:
 - Decision tables, equivalence partitioning, boundary value analysis

 NPTEL

Along with this we also saw lot of the classical terms and testing that you would encounter in any old test textbooks. Related to source code, these are the terminologies

that I introduced you to: cyclomatic complexity, independent paths, basis path, testing decision to decision paths. I also told you in the context of white box coverage, statement coverage, branch coverage, loop coverage. We saw them as node coverage edge coverage and prime path coverage. Then when we focusing on design integration testing, we learnt about interfaces, types of interfaces, message passing, shared variable and so on. We learnt about what stubs and drivers are when it comes to integration testing. We learnt various approaches to integration testing: incremental, top down, bottom up, big bang and sandwich. When it comes to black box testing techniques, we learnt about equivalence partitioning, decision tables and boundary value analysis.

(Refer Slide Time: 12:17)



Formal languages

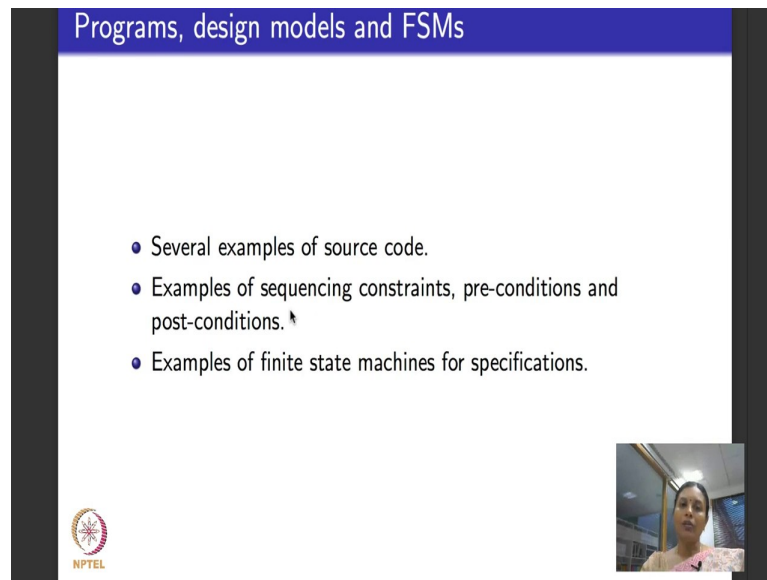
Presented an overview of

- Propositional logic, basics of predicate logic.
- Finite state automata.
- Regular expressions.
- Context-free grammars.

NPTEL

In this process of lecturing to you about test phase design and criteria we also saw a good number of formal models and languages. We saw the basics of propositional logic and predicate logic. I introduced you to finite state automata, we saw regular expressions we saw context free grammars.


(Refer Slide Time: 12:41)



Programs, design models and FSMs

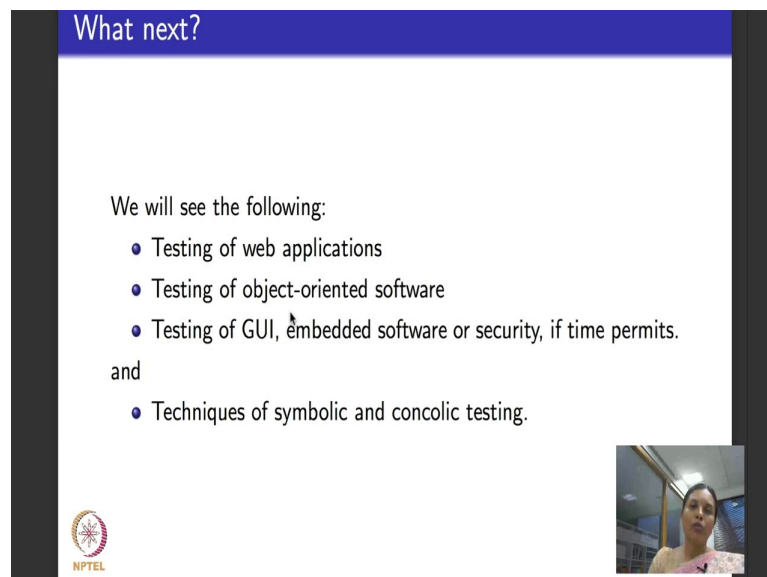
- Several examples of source code.
- Examples of sequencing constraints, pre-conditions and post-conditions.
- Examples of finite state machines for specifications.

NPTEL



Of course, not very rigorously at a shallow level, but we did introduce them formally and saw results and examples about them and we saw several examples of source code, several examples of design constraints like sequencing constraints, preconditions, post conditions. We saw examples of finite state machines for specifications and XML.

(Refer Slide Time: 12:54)



What next?

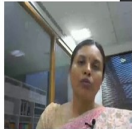
We will see the following:

- Testing of web applications
- Testing of object-oriented software
- Testing of GUI, embedded software or security, if time permits.

and

- Techniques of symbolic and concolic testing.

NPTEL



What are we going to do now from week 10 onwards for the remaining 3 weeks. I will introduce you to testing of web applications, object oriented applications wherever applicable will come back and look at how the coverage criteria we learnt applies here. I

will also introduce you if time permits to testing of embedded software, security testing, and about GUI testing. I am not confident at this stage that I will be able to cover all of them we will try to do as much as possible, but whatever we do, we will do it rigorously and thoroughly. And I would like to spend some time in the last couple of weeks doing these two techniques which are called symbolic testing or concolic testing which have been there for the past 30-40 years, but they have surfaced now in a big way and they give you a very niche set of algorithms to do path based coverage in testing and they also do the instrumentation for you.

So, this is where we stand and this is what we are going to do. I hope this overview lecture was useful for you to sit back and reflect where we are in the course, what we did till now and what are we going to do from now on. So, next week when I meet you the first video will be on testing of web applications.

Thank you.

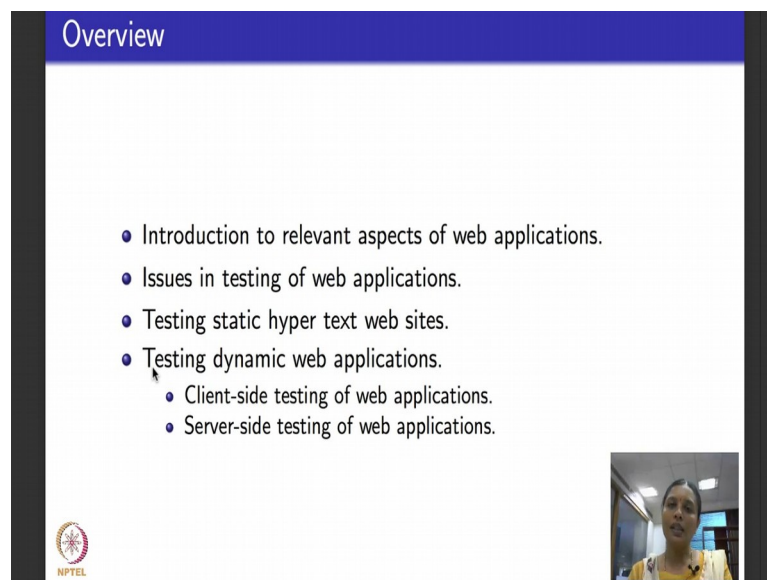
Software testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 45
Testing of Web Applications and Web Services

Hello again, welcome to week 10. We are going to start on something completely new this week. As I told you towards the end of the last lecture of week 9 we saw what we did till now we mainly did coverage criteria and so various kinds of testing using this model or that structure--- graphs, logic, sets and so on. This week I am going to take classes of applications we are going to start with web applications web software move on to object oriented applications, we learn a bit about those applications, we learn problems that are unique related to testing about each of these applications. We will compare and see the coverage criteria that we have learnt so far whether they can be directly used to test these applications or not and if not I will tell you new algorithms that can be specifically used to test applications.

So, we are going to begin this week by looking at testing of web applications.

(Refer Slide Time: 01:05)



Overview

- Introduction to relevant aspects of web applications.
- Issues in testing of web applications.
- Testing static hyper text web sites.
- Testing dynamic web applications.
 - Client-side testing of web applications.
 - Server-side testing of web applications.

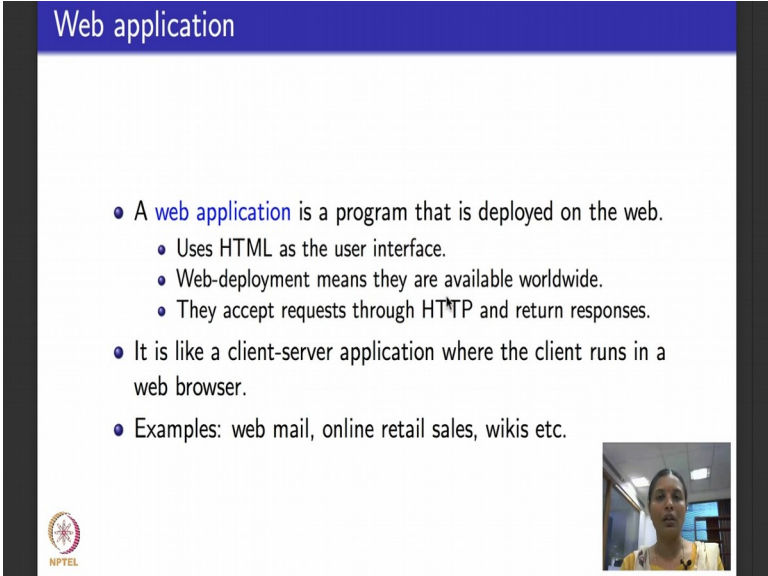
NPTEL

So, this is an overview of what we are going to do over the next few lectures. Today I will introduce you to relevant aspects of web applications we will see what they are and

then we will also discuss what are the specific issues that come with testing of web applications.

And then when it comes to testing it helps to have the following broad categorization we will test what are called static hypertext websites, websites that remain the same, look the same to almost every user who accesses it all the time. Basically have nothing more than HTML files. And then when it comes to testing its important to separately consider dynamic web applications. In dynamic web applications the content of the web page is created dynamically by an underlying application or a program. Here usually the kind of code that is written for the server side is very different from the kind of code that is done for the client side. Hence the testing that we have to do for the client side and server side are also very different. So, this is will, this is a structure of what we are going to see about testing of web applications put together.

(Refer Slide Time: 02:12)



The slide is titled "Web application" in a blue header. It contains a list of bullet points defining web applications. In the bottom right corner, there is a small video inset showing a woman speaking. The NPTEL logo is in the bottom left corner.

- A **web application** is a program that is deployed on the web.
 - Uses HTML as the user interface.
 - Web-deployment means they are available worldwide.
 - They accept requests through HTTP and return responses.
- It is like a client-server application where the client runs in a web browser.
- Examples: web mail, online retail sales, wikis etc.


Today, I will begin with introducing you to the relevant aspects of web applications. Please remember that this is not meant to be a comprehensive introduction to what a web application or a web software is. Feel free to look up other courses or online material for that. I will just give you a bird's eye overview of what it is and consider it as understood from the point of view of testing. Our issue will be to focus on testing not to thoroughly introduce you to web applications.

So, what is a web application for us? For us a web application is a program written in some language right, but the program, difference is not deployed in the computer it is deployed on the web. What do we mean by its deployed on the web? It is basically deployed in a computer that is meant to act as a server for further deployment or availability on the web or the internet. What is the user interface that this kind of web application uses its basically HTML stands for hypertext markup language, standard language that people use to write web pages. Web deployment means what, its available on the internet worldwide as long as you have access to it, you have access to this website.

And typically a web application, how does it talk to a user, an end user? It talks to an end user through a server that accepts requests from an end user or a client through HTTP - HTTP stands for hypertext transfer protocol. It is a standard protocol that is used in the internet and it also returns responses which would websites basically your information provided by a web page through HTTP again. So, this is a web application for us. So, it is a lot like a classical client server application, the only difference is the server is a web server meant to host the application on the internet and the client runs in a web browser. What a web browsers? Internet explorer, Mozilla, Google Chrome and so on.

So, here are examples of classical web applications. We all use web mail some form of the other--- Gmail, Microsoft office outlook, yahoo mail, hotmail and so on. We also use a lot of online retail sales--- amazon, flipkart and so on and we use wikis these are various categories of web applications and lots and lots more are also there.

(Refer Slide Time: 04:28)



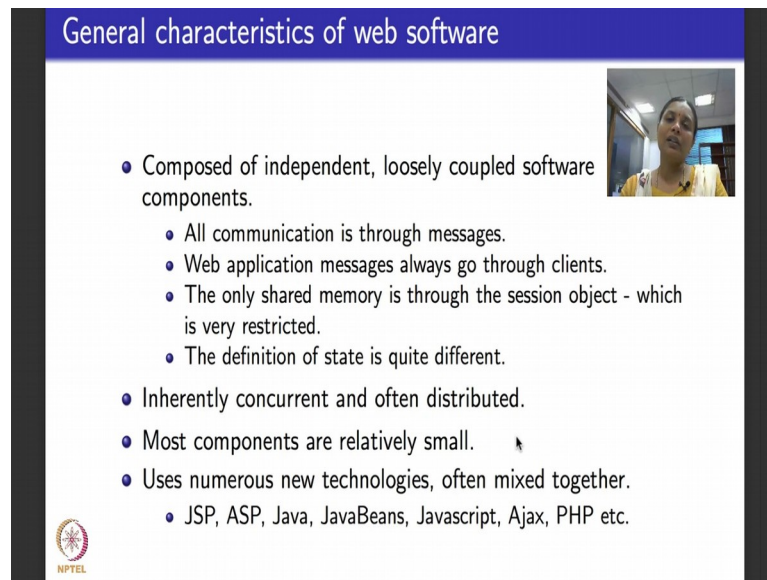
The slide has a blue header with the text "Web services". Below the header, there is a bulleted list defining a web service. In the bottom right corner, there is a small video inset showing a person. The NPTEL logo is in the bottom left corner.

- A **web service** is a web-deployed program that accepts XML messages wrapped in SOAP.
 - Usually no user interface with humans.
 - Service must be published so other services and applications can discover them.

There are a slightly special class of web applications called web services. What is a web service? Web service like a web application is another program that is deployed on the web, it accepts XML messages that are wrapped in SOAP or restful services. Usually it has no interface with human it directly talks to programs or clients and the service must be published so that other services and applications can discover a web service.

In our lectures when we focus on testing we will not really explicitly consider web services as a standalone separate kind of web application, we look at generically testing web applications as we introduce them here.

(Refer Slide Time: 05:10)



General characteristics of web software

- Composed of independent, loosely coupled software components.
 - All communication is through messages.
 - Web application messages always go through clients.
 - The only shared memory is through the session object - which is very restricted.
 - The definition of state is quite different.
- Inherently concurrent and often distributed.
- Most components are relatively small.
- Uses numerous new technologies, often mixed together.
 - JSP, ASP, Java, JavaBeans, Javascript, Ajax, PHP etc.

NPTEL

So, what are the general characteristics of such web applications or web software? I will use the term web app, web application, web software all these terms synonymously, they all mean the same for us. Web application is composed of several independent loosely coupled program components.

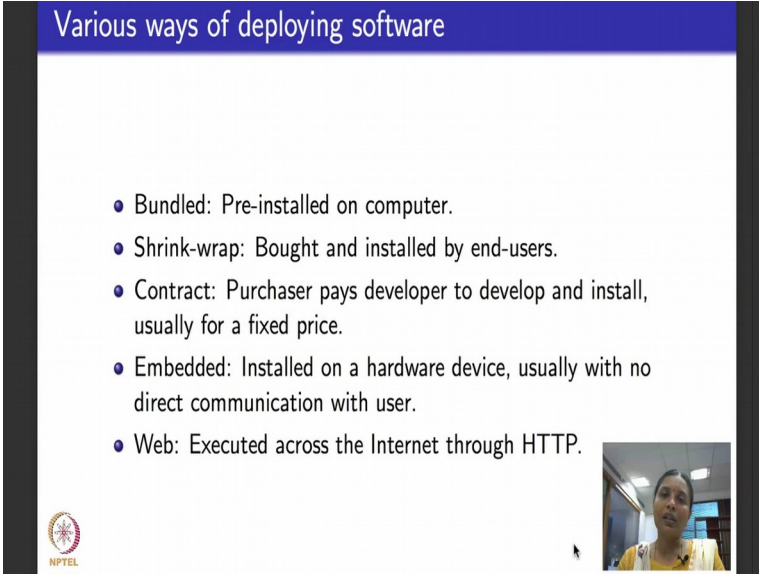
How do these loosely coupled program components talk to each other? How do they constitute one web application? They communicate with each other through message passing, the interface that they use is message passing interface and these messages from the application always go through dedicated entities called clients and sometimes they do share memory, but it is very restricted it is through what is called session objects and the definition of a state of a program we know what it is. It means the class of all set of all variables of a program and a location counter, but here the state of a program as a web application is usually quite different, does not correspond to the classical notion of a program state.

As I told you because it consists of independent loosely coupled components a web application is a distributed or a concurrent application and it has several components each of them very small and not only that each of these components could be developed or written processed using a completely different technology. For example, somebody could do JSP Java server pages somebody could do ASP, somebody could just write

Java, Javabeans, Javascript, Ajax, PHP, HTML. If you go and search for web technologies and web applications you will find all these terms through and across.

My goal again, to reiterate, is to not to exhaustively introduce you to these terms, but to more bring it up as one of the characteristics of web applications. That when we put them all together to test we have to handle in our tests several different technologies that come from completely different domains, that is important to remember.

(Refer Slide Time: 07:09)



The slide is titled "Various ways of deploying software" in a blue header. It contains a bulleted list of five deployment methods. In the bottom right corner, there is a small video inset showing a person speaking. The NPTEL logo is in the bottom left corner.

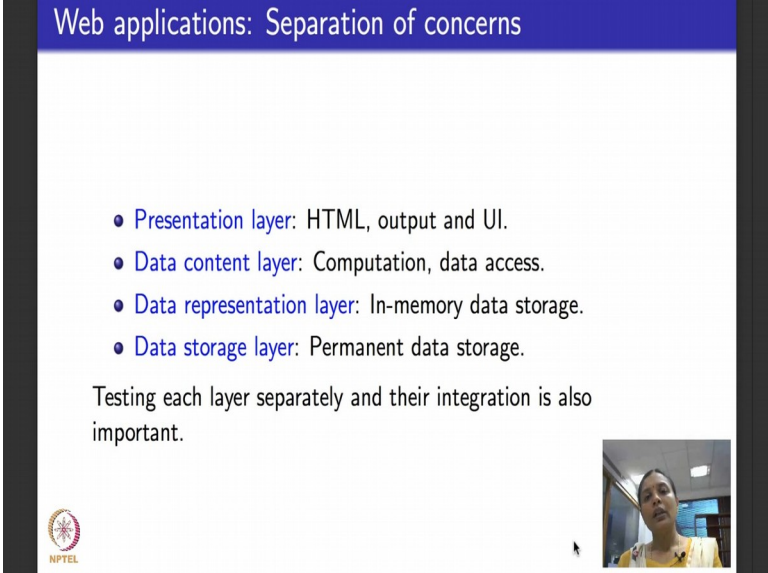
- Bundled: Pre-installed on computer.
- Shrink-wrap: Bought and installed by end-users.
- Contract: Purchaser pays developer to develop and install, usually for a fixed price.
- Embedded: Installed on a hardware device, usually with no direct communication with user.
- Web: Executed across the Internet through HTTP.

Typically when I consider a software any piece of software that you have what are the various ways of deploying it? This is not specific to web applications or web software, any generic software you could bundle a software in which case its usually pre installed on a computer.

Like for example, if you buy a package, if you buy a computer then the operating systems in the computer comes bundled along with it its pre installed the way you want it. You could shrink wrap a typical example for this is you buy the software and install it like an anti-virus software that you would buy and install. Or you would have a contract software where you have access to the software, but for a specified period of time that you buy the contract for and you could have embedded software which is usually installed on a specific purpose special purpose hardware device like in a microwave oven in a car, in an aeroplane and so on. And it typically directly communicates only with sensors and actuators and the embedded device, does not communicate with the user.

And then now the focus of this lecture is what is called web software. How is it deployed? It is executed across the internet through HTTP protocol typically hosted by a dedicated server for this purpose called the web server.

(Refer Slide Time: 08:22)



Web applications: Separation of concerns

- Presentation layer: HTML, output and UI.
- Data content layer: Computation, data access.
- Data representation layer: In-memory data storage.
- Data storage layer: Permanent data storage.

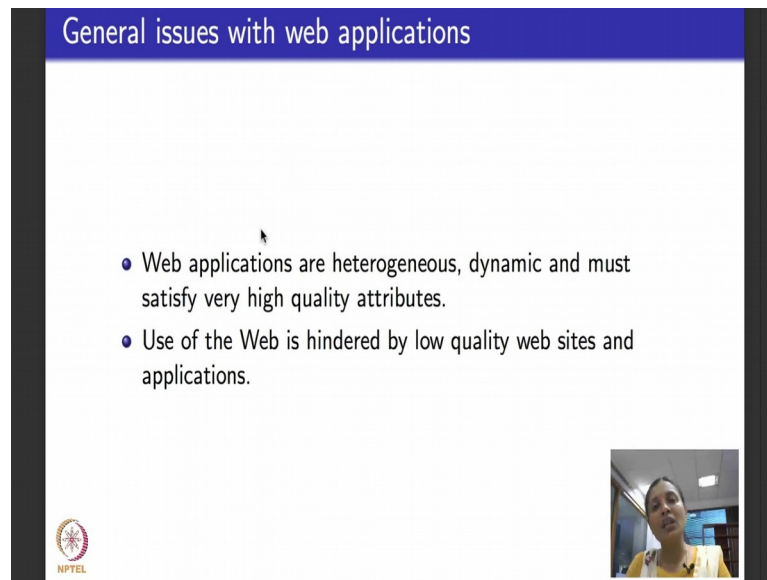
Testing each layer separately and their integration is also important.

NPTEL

Now, let us look at web application, consider the typical architecture of a web application. The architecture has several layers where separation of concerns happen. There is something called the higher, one of the higher layers of the architecture is a presentation layer where the HTML and UI output are visible. Then comes the layer that contains the data called the data content layer where the actual computation and access to data happen.

Below the data content layer is a data representation layer where the data is stored and retrieved in memory and finally, you have data storage layer which represents permanent data storage that happens because of a web application. Testing each layer separately is important and what is also important is to do system level testing, where you test for integration across these layers that do separation of concerns in a web application. Here are some generic issues with web application? They, as I told you, are heterogeneous dynamic distributed and because they are available freely across the internet typical expectation is that they must be of very high quality. They cannot be slow, they cannot have a bug, they have to be reliable, they need to respond very quickly.

(Refer Slide Time: 09:16)



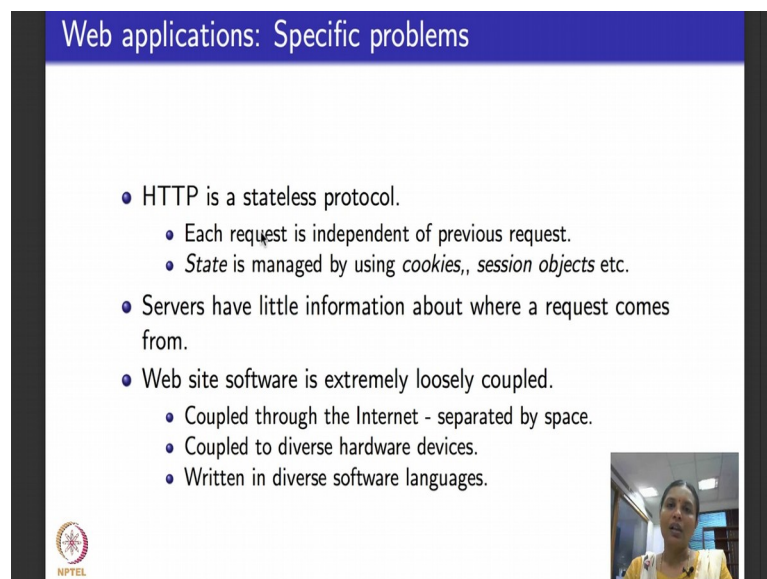
General issues with web applications

- Web applications are heterogeneous, dynamic and must satisfy very high quality attributes.
- Use of the Web is hindered by low quality web sites and applications.

The slide features a blue header with the title 'General issues with web applications'. Below the title, there are two bullet points. In the bottom right corner, there is a small video inset showing a woman speaking. The NPTEL logo is visible in the bottom left corner.

So, their quality attributes and expectations from the software is very high, but you while know by experience that there are so many low level software that actually hinder the use of these web applications.

(Refer Slide Time: 09:49)



Web applications: Specific problems

- HTTP is a stateless protocol.
 - Each request is independent of previous request.
 - State is managed by using *cookies*, *session objects* etc.
- Servers have little information about where a request comes from.
- Web site software is extremely loosely coupled.
 - Coupled through the Internet - separated by space.
 - Coupled to diverse hardware devices.
 - Written in diverse software languages.

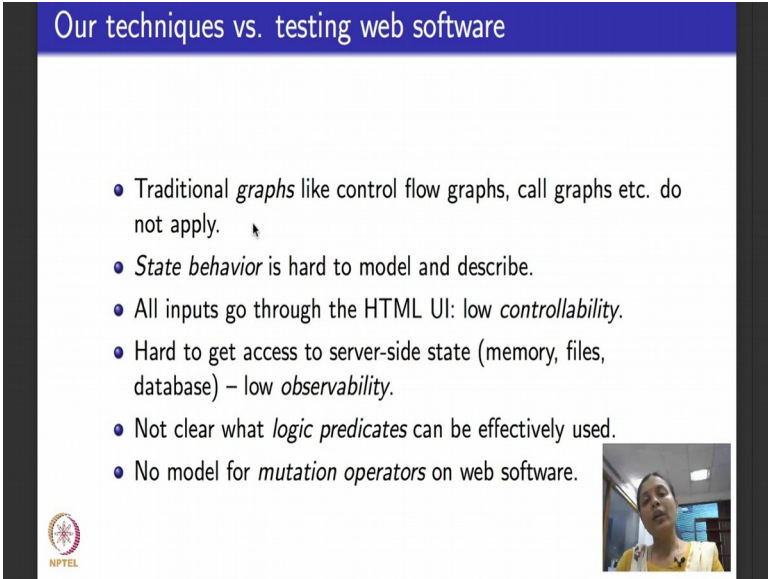
The slide features a blue header with the title 'Web applications: Specific problems'. Below the title, there are four bullet points. The second bullet point has a sub-list. In the bottom right corner, there is a small video inset showing a woman speaking. The NPTEL logo is visible in the bottom left corner.

Now, let us move on and discuss specific issues related to web applications. As I told you all web applications and web software use hypertext transfer protocol HTTP. What do we know about HTTP, it is a stateless protocol.

What do we mean by it is a stateless protocol? Suppose there is a new HTTP request it is completely independent of the history. It is completely independent of the previous requests, the requests before that and so on. So, if it is a stateless protocol how do people manage state when they deploy web application? You all might have heard of the term cookies, session objects and so on right. So, those are basically entities that web programmers or web app developers use to manage state information whenever needed.

The next is, web server typically has very little information about where a request comes from. It could be from malicious client, it could be from a genuine client. Typically you do lot of authentication and authorization, but still there is very little information about where exactly the request is coming from and what happens to the response in turn. Web software as we discussed earlier its extremely loosely coupled right, extremely loosely, there is importance to that word why, there is something called tightly coupled and loosely coupled here I have one more qualifying adjective, I say it is extremely loosely coupled. Why is that so, because it is coupled through the internet which means what the software is physically separated in space. It does not reside in one entity, it does not reside within one server. It is also coupled to diverse hardware devices, it could be anywhere in any part of the world and as I told you its written using diverse technologies and a diverse set of software languages.

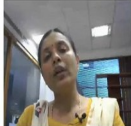
(Refer Slide Time: 11:36)



Our techniques vs. testing web software

- Traditional *graphs* like control flow graphs, call graphs etc. do not apply.
- *State behavior* is hard to model and describe.
- All inputs go through the HTML UI: low *controllability*.
- Hard to get access to server-side state (memory, files, database) – low *observability*.
- Not clear what *logic predicates* can be effectively used.
- No model for *mutation operators* on web software.

NPTEL



Let us look at all the techniques that we have learnt so far and analyze one at a time about whether any of them will be useful for testing web software. What did we begin the course with? We taught graphs to start with, but what were the graph models of software artifacts that we considered? We considered control flow graph, we considered call graphs and so on. If you see none of these kind of models apply for system level testing of web application.

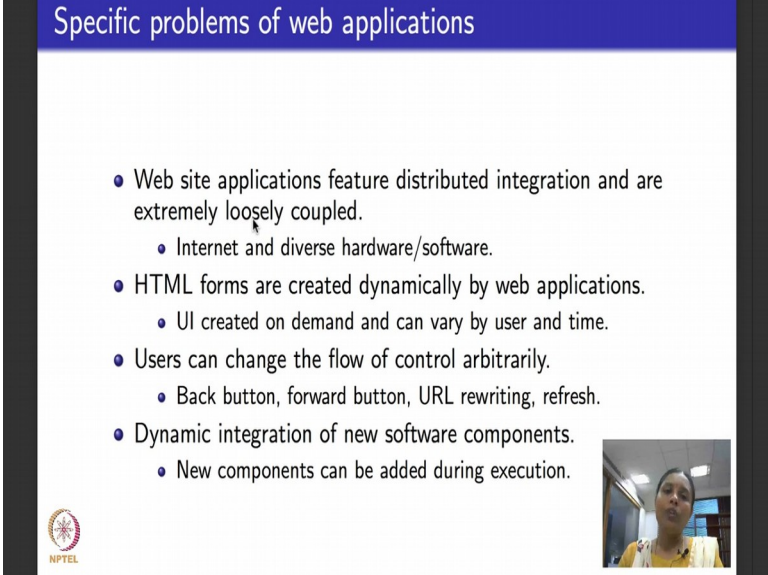
Of course you could consider one individual program that is written as a web application and if you are doing unit testing for that program then you could consider the control flow graph do your coverage criteria. But that is not what we talking about, we talking about system level testing of a web app. In that case working with control graph or call graph does not really apply because the program is distributed across the internet. The other thing is, we also learnt about state machines, the specific graph models that model the state behavior of software. Here state behavior of a web application, if you think about it, there is some part of the server state, there is some part of the client state, different clients have different states. It is quite a hard task to model the state.

Typically all, the next important issues are related to RIPR model or controllability and observability. Typically both controllability and observability for web software is very low. Why is controllability low, because all inputs please remember go through HTML files right. So, if I have to control an input I am talking about controlling the entire HTML file of an entire webpage. It is going to be difficult to provide inputs that can be controlled within a user, and once you provide an input, let us say you manage to provide an input, you managed to get controllability, the next is to observe what happens in this for the state of a web program which is hosted in a web server. It is very difficult to get access to server side details. For good reasons it is not secure enough and typically do not want to, we do not know where the memories are, where the file is where the database is. So, from the point of view of testing observability and controllability becomes very low, so testing becomes difficult.

Now the other kind of software artifact that we learnt were logic predicates, but it is not clear what kind of logic predicates it can be effectively used to do system level testing. Again the last one that we learnt was related to mutation operators. To the extent that I am aware of there are no known mutation operator specifically for web software. So, it looks like most of the techniques that we learnt cannot be directly put to use. We will use

graph based testing to some extent, but only for restricted websites that are static in nature. The rest of them we really have to understand or develop pretty new techniques that could be based on the techniques that we learnt so far, but none of the ones that we have learnt so far directly apply.

(Refer Slide Time: 14:35)



The slide is titled "Specific problems of web applications" in a blue header. It contains a bulleted list of five points, each with a sub-bullet. In the bottom left corner is the NPTEL logo, and in the bottom right corner is a small video inset showing a person speaking.

- Web site applications feature distributed integration and are extremely loosely coupled.
 - Internet and diverse hardware/software.
- HTML forms are created dynamically by web applications.
 - UI created on demand and can vary by user and time.
- Users can change the flow of control arbitrarily.
 - Back button, forward button, URL rewriting, refresh.
- Dynamic integration of new software components.
 - New components can be added during execution.

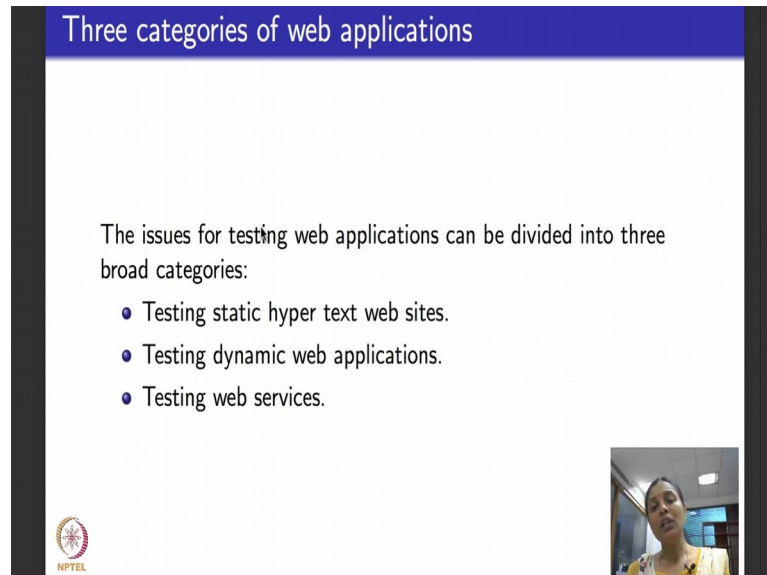
So, now we will revisit the specific problems of web applications because it is useful to know when we are designing testing techniques. So, as I told you they are very extremely loosely coupled, over the internet, this diverse hardware and software and I told you HTML forms are created dynamically by web applications.

So, there is very little controllability and observability on the input. The user interface that is provided through a HTML file can vary and it is created on demand based on the credentials, and users can change the flow of control arbitrarily in the middle of a transaction as a page is loading somebody could press a back button or a forward button could press a refresh button. That is why when you handle secure transactions like bank and that you explicitly get messages saying do not press back button. But typically for a normal web application there is no expectation that nobody will press the back buttons.

So, as your testing while your input is executing, while your program is running if the control changes then what you observe may not actually correspond to the actual web app or the web software behavior at all. So, it is quite a dicey thing. And the other thing

is that dynamically new software components can be added into a web application. So, all these are issues.

(Refer Slide Time: 15:53)



The slide is titled "Three categories of web applications" in a blue header. The main content area is white and contains the text: "The issues for testing web applications can be divided into three broad categories:" followed by a bulleted list: "• Testing static hyper text web sites.", "• Testing dynamic web applications.", and "• Testing web services." In the bottom left corner, there is a small NPTEL logo. In the bottom right corner, there is a small video inset showing a woman speaking.

Three categories of web applications

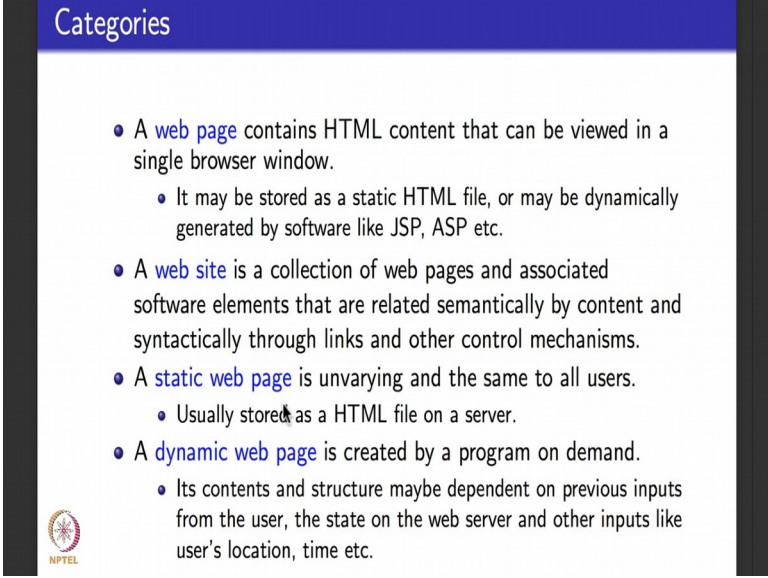
The issues for testing web applications can be divided into three broad categories:

- Testing static hyper text web sites.
- Testing dynamic web applications.
- Testing web services.

NPTEL


When it comes to testing we prefer to bucket web applications into three different categories. We will separately deal with what we call static hypertext websites which are basically plain HTML files that are written once, put on the web server deployed and not touched. And then will separately deal with testing dynamic web applications. On that will do separately client side testing and separately server side testing. As I told you we really will not exclusively consider testing of web services. We will generically deal with testing of web applications. The last topic here we will not specifically consider it.

(Refer Slide Time: 16:33)



Categories

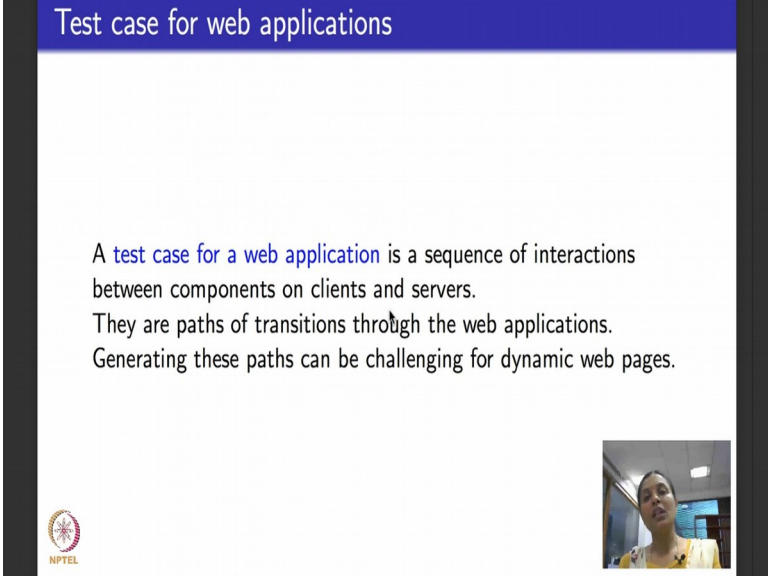
- A **web page** contains HTML content that can be viewed in a single browser window.
 - It may be stored as a static HTML file, or may be dynamically generated by software like JSP, ASP etc.
- A **web site** is a collection of web pages and associated software elements that are related semantically by content and syntactically through links and other control mechanisms.
- A **static web page** is unvarying and the same to all users.
 - Usually stored as a HTML file on a server.
- A **dynamic web page** is created by a program on demand.
 - Its contents and structure maybe dependent on previous inputs from the user, the state on the web server and other inputs like user's location, time etc.



So, just to re-define find a few terminologies that we will need for the rest of the few lectures. For us a web page contains HTML content and it is typically viewed in a browser single browser window. It can be a static HTML file or it could be dynamically generated by using JSP, ASP or any other kind of web software.

A web site is a collection of web pages, static and or dynamic and its associated software element that are related semantically in a meaningful way by their content. How are they related syntactically? They are related syntactically by hyperlinks and other mechanisms for passing control from one web page to the other--- hyperlinks forms and so on. A static web page never changes, it is the same to all the users all the time as long as it is available, usually stored as a HTML file on a dedicated web server. In contrast, a dynamic web page is created by a program or a web program web software on demand, it contents and structure typically depend on previous inputs from the user, state of the web server, other inputs like where the user is from, what time of the day, what is the data provided by the user and so on.

(Refer Slide Time: 17:53)



The slide has a blue header with the text "Test case for web applications". The main content area is white and contains the following text:

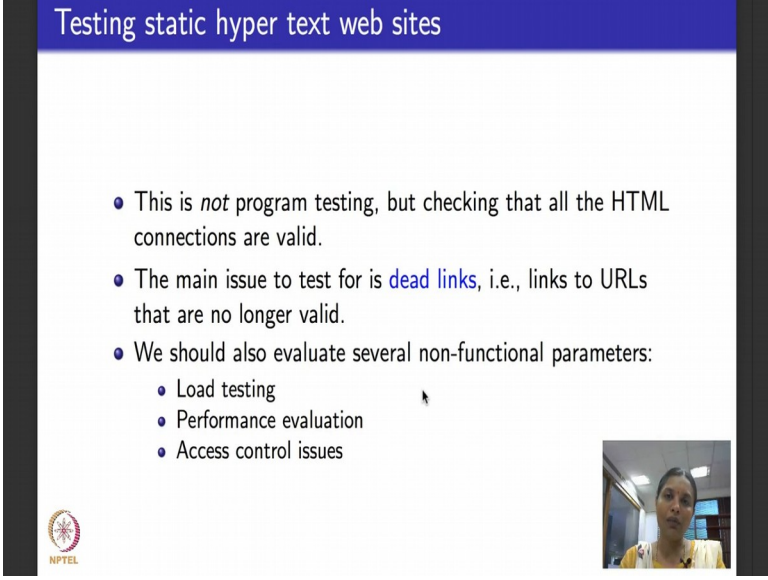
A test case for a web application is a sequence of interactions between components on clients and servers.
They are paths of transitions through the web applications.
Generating these paths can be challenging for dynamic web pages.

In the bottom right corner, there is a small video inset showing a person speaking. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.

For us when we test web applications, it helps to first think back and understand what will a test case for a web application be. A test case for a graph will be a path, a test case for a logical predicate will be true or false values to the inputs. A test case for mutation based testing would be any value that kills the mutant and so on. Similarly what is a test case for a web application? A test case for a web application is basically a sequence of interactions between components that reside on the clients and the components that reside on the server. How are they given? Test case is usually depicted by a path that contains states and transitions through the various web applications. These paths have to be generated for static and dynamic websites. For static web pages, it is easy to do. For dynamic websites it can be challenging mainly because as I told you, you could have users changing the flow of control arbitrarily. So, if you generate a particular dynamic web content, halfway through the content being fed into the software as an input, the nature of the software itself could be changed by the user.

So, just as a recap what is the test case a test case is basically a sequence of web pages which are given as transitions through the web applications; easier to create for static websites little more difficult to create for dynamic websites.

(Refer Slide Time: 19:24)



The slide is titled "Testing static hyper text web sites" in a blue header. It contains a bulleted list of testing points. The first point states that this is not program testing but checking for valid HTML connections. The second point mentions testing for dead links. The third point lists non-functional parameters: load testing, performance evaluation, and access control issues. An NPTEL logo is in the bottom left, and a small video inset of a woman is in the bottom right.

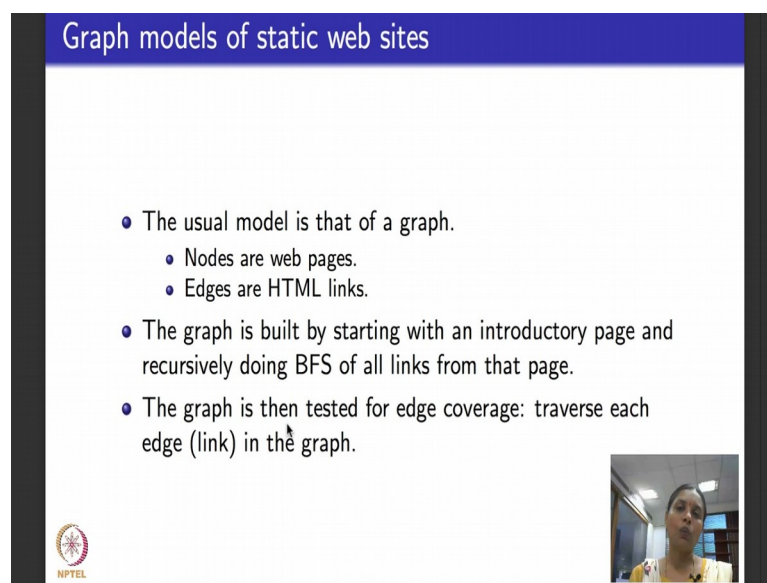
- This is *not* program testing, but checking that all the HTML connections are valid.
- The main issue to test for is **dead links**, i.e., links to URLs that are no longer valid.
- We should also evaluate several non-functional parameters:
 - Load testing
 - Performance evaluation
 - Access control issues

So, we will begin by looking static websites in this lecture. Next lecture I will begin by doing dynamic websites. Since static website what is a web page look like there is no program involved at all please remember that, maybe there is some scripting involved the ultimate contents to test is a set of plain HTML files that never fall. This is not program testing because there is no program. But what is this testing do? It basically checks that all the HTML connections are valid right. If one hyperlinks another one that connection is valid and it goes to the intended page, not only does it go to the intended page, you also check that the particular hyperlink or a connection that is there available from a website does not go to any URL that is dead or defunct or that are no longer valid.

So, such kind of testing should also include testing for what are called dead links. Dead links are those that are links, hyperlinks to URLs that are not valid. Typically when it comes to system level testing of web applications people also evaluate several non functional parameters. They test the load of a system, classical example would be let us say use IRCTC web application, load testing would be to let us exercise maximum number of users that the system can scale up to. Let everybody log in and see if the system is still able to smoothly let the person transact and book his or her ticket the way he or she wants. Another thing is performance evaluation which basically talks about the fast, the efficiency of response to of a web application.

The third is access control issues in the sense that you need certain kind of accesses to be able to access certain kind of dynamic web page. Like for example, if you are accessing your bank webpage before you actually get down to the dynamically created content that gives you your user account details, your access as a user needs to be authenticated by means of a login and a password. So, system level testing of web applications also include these non functional parameters. Our focus in these lectures would be on functional testing of system level parameters. So we will not really look at load or performance testing in this course.

(Refer Slide Time: 21:41)



The slide is titled "Graph models of static web sites" in a blue header. It contains three bullet points:

- The usual model is that of a graph.
 - Nodes are web pages.
 - Edges are HTML links.
- The graph is built by starting with an introductory page and recursively doing BFS of all links from that page.
- The graph is then tested for edge coverage: traverse each edge (link) in the graph.

In the bottom left corner is the NPTEL logo. In the bottom right corner is a small video inset showing a woman speaking.

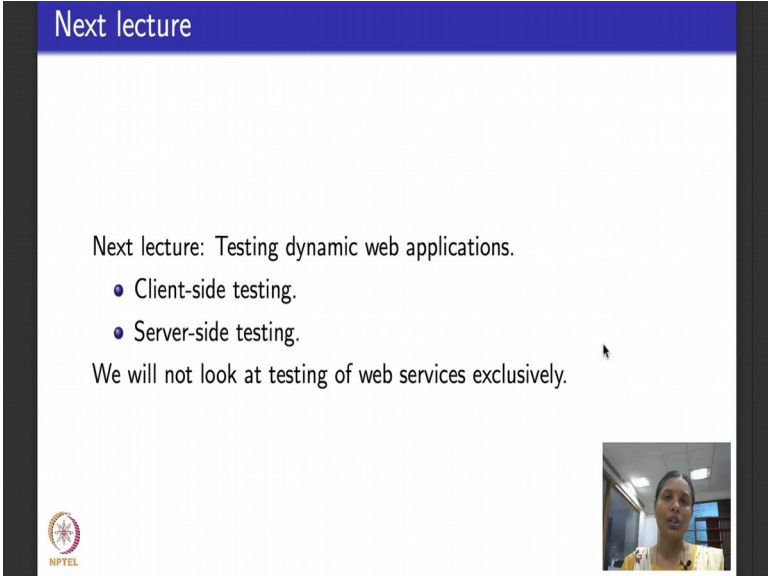
So, when it comes to static website what is the graph, what is the model of testing that we use? It is not too difficult to see that graph models as we saw will always work. Of course, this is not call graph or a control flow graph, this is a different kind of graph. What will be the nodes or vertices of this graph? The nodes or vertices of these graphs are the various web pages, each individual page. What are the edges of these graphs? Let us say you are in one particular page, there is a hyperlink, you click on that hyperlink you go to another page. Then you say that there is an edge from the node corresponding to the first webpage to the node corresponding to the second web page. So, edges are HTML links.

And how do you build such a graph? You typically build the graph by starting from an introductory page. In static pages they will usually be a index dot HTML or a welcome

dot HTML which is the beginning page of the static website. You start from there do a breadth first search, recursively, by looking at each link from that page and then go to the pages that it links to, again do a BFS to the links of that page and so on till there are no further links to be explored. Once this graph is generated what is the main aspect that is tested? The main aspect that is tested is classical edge coverage. What do edges of this graph represent? Is each link working fine?

So, edge coverage basically tests for traverse each hyperlink in the static web site. So, testing static web sites, that is about it there is nothing much. You generate a graph vertices of the graph are web pages, edges of the graphs are hyperlinks and then you do edge coverage in the graph. This basically tests for static website working correctly, because there is nothing like performance load, there is no dynamic data. That is it we are done with system level testing of static websites.

(Refer Slide Time: 23:29)

A presentation slide titled "Next lecture" in a blue header bar. The slide content includes the text "Next lecture: Testing dynamic web applications." followed by a bulleted list with two items: "Client-side testing." and "Server-side testing." Below the list, it says "We will not look at testing of web services exclusively." In the bottom left corner is the NPTEL logo, and in the bottom right corner is a small video inset showing a woman speaking.

Next lecture

Next lecture: Testing dynamic web applications.

- Client-side testing.
- Server-side testing.

We will not look at testing of web services exclusively.

NPTEL

Now, the next lecture what I will tell you is let us move on and look at dynamic web applications. When it comes to dynamic web applications, as far as testing is concerned as I told you it helps to distinguish between the client side testing and server side testing. So, we look at these two independently and see some new techniques or technologies or algorithms that have been developed for client side testing and for server side testing and as I told you a couple of times before in this lecture, we will not look at testing web services exclusively.

So, I will come back to you in the next lecture for testing a dynamic websites.

Thank you.

Software testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 46
Testing of Web Applications and Web Services

Hello again, welcome to the second lecture of week 10. If you remember we had started looking at web applications testing in the last lecture.

(Refer Slide Time: 00:20)



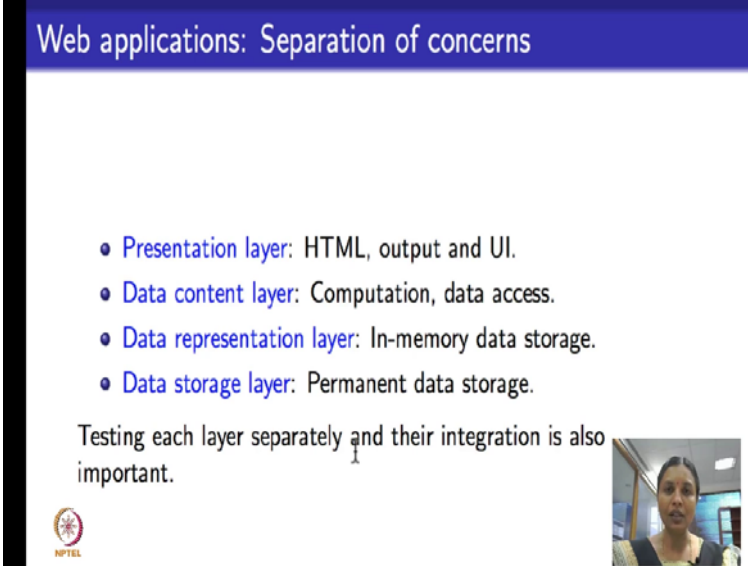
I had given you the following overview. So, what I did in the last lecture was to introduce you to relevant aspects of web applications it by no means has comprehensive introduction of web application, but as we would need it from the point of view or testing and criteria based testing we saw the basics. And I also talked about the issues that typically come up while testing web applications and then we ended the lecture with the brief mention of how to test static HTML, plain HTML web sites.

You basically model it as a graph where the nodes of HTML pages and the edges are hyperlinks and then that would check for typical link be working fine dead links and so on. What we going to do in the next two lectures is to look at dynamic web applications which is the last bullet her. Dynamic web applications typically have a web server as we saw in the last class where all the software of the web application reside, client access the software in the web server and the web page that the client is accessing from the server

gets dynamically generated based on how the client interacts with the software residing in the server.

So, you could test this by testing on the client side of the web application or by testing on the server side of the web application. The code on the client side basically talks about the script validation that happen at the level of client. The code on the server side is fairly big large complicated web software. What we will do in this module is we will understand how to do clients side testing of web applications. In the next module I will tell you how to do server side testing of dynamic web applications.



(Refer Slide Time: 02:08)



Web applications: Separation of concerns

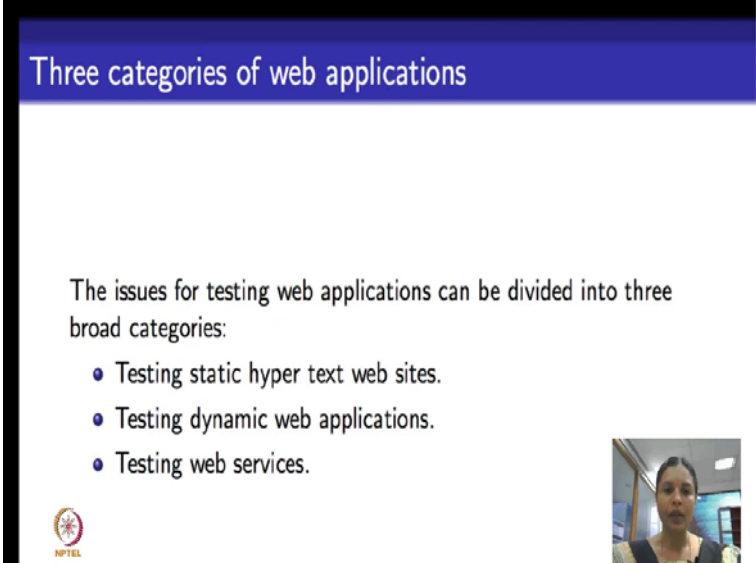
- **Presentation layer:** HTML, output and UI.
- **Data content layer:** Computation, data access.
- **Data representation layer:** In-memory data storage.
- **Data storage layer:** Permanent data storage.

Testing each layer separately and their integration is also important.



And using the same slide deck continuing with the same slide deck that I did, so I will skip past this slides except for recapping whatever is needed. Please remember that web applications, architecture of web applications typically at a high level I have this several layers in which software is present. Each layer is typically separately tested their integration is also tested for a large fragment of the testing that we are going to see in this module and next module for web applications, we will stick to presentation layer may be I will briefly touch upon the data content layer. The software that we will be testing as a web application mainly resides only in the presentation layer. For client side and server side testing we will be consider the software as it is present in the presentation layer.

(Refer Slide Time: 02:59)



The issues for testing web applications can be divided into three broad categories:

- Testing static hyper text web sites.
- Testing dynamic web applications.
- Testing web services.

The slide features a blue header with the title 'Three categories of web applications'. In the bottom left corner, there is a small circular logo with a star and the text 'NPTEL' below it. In the bottom right corner, there is a small video inset showing a woman speaking.

So, I am skipping past the rest of this slide which we already discussed in the last lecture. So, as I told you that issue of testing web application is broadly divided into three categories testing static web sites, testing dynamic web applications where we will see server and client separately and then testing web services which we are going to skip.

(Refer Slide Time: 03:22)

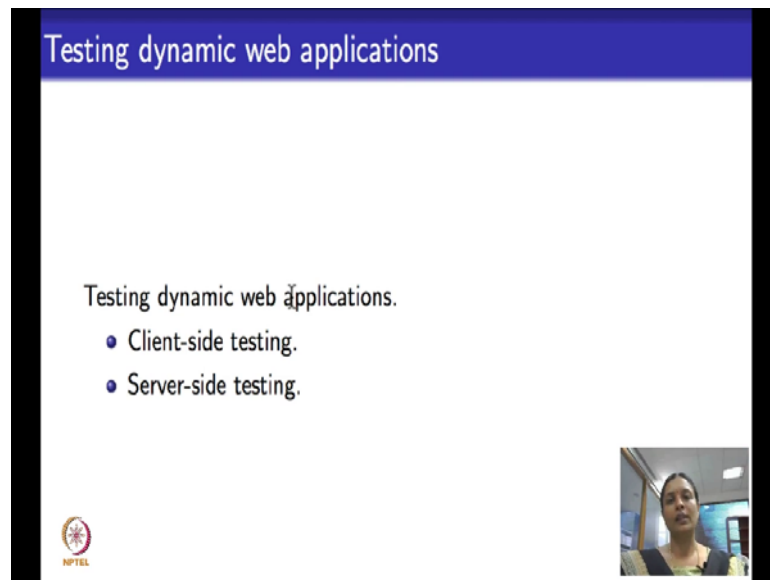


A **test case for a web application** is a sequence of interactions between components on clients and servers. They are paths of transitions through the web applications. Generating these paths can be challenging for dynamic web pages.

The slide features a blue header with the title 'Test case for web applications'. In the bottom left corner, there is a small circular logo with a star and the text 'NPTEL' below it. In the bottom right corner, there is a small video inset showing a woman speaking.

Last module I introduce you to a web page a static web page a dynamic web page and how a test case looks like. It is basically a sequence of interactions between components software components that lie on clients or servers.

(Refer Slide Time: 03:43)



Testing dynamic web applications

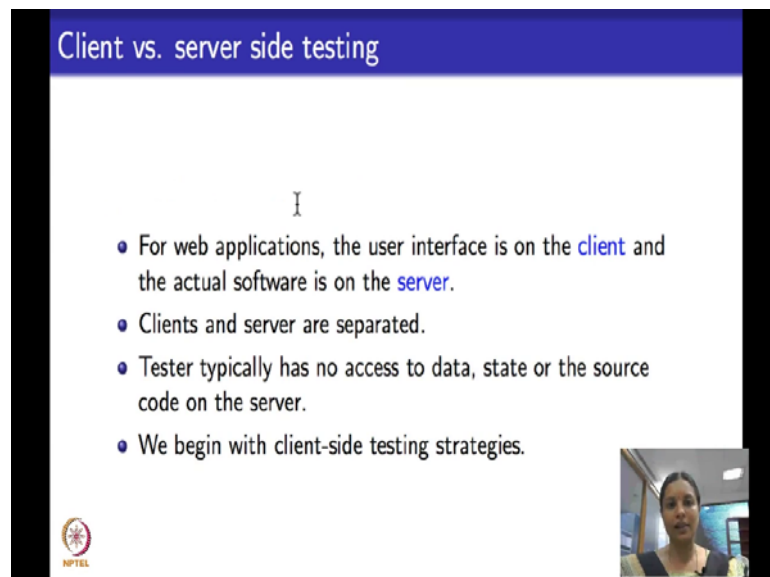
Testing dynamic web applications.

- Client-side testing.
- Server-side testing.

The slide features a blue header with the title 'Testing dynamic web applications'. Below the title, the text 'Testing dynamic web applications.' is followed by a bulleted list containing 'Client-side testing.' and 'Server-side testing.'. In the bottom left corner, there is a small circular logo with the text 'NPTEL' underneath it. In the bottom right corner, there is a small video inset showing a woman with dark hair, wearing a yellow top, speaking.

And we also saw how to test for static web pages has graphs. Now we going to begin with this. We are going to test dynamic web applications. Separately we going to see two things: client side testing of dynamic web applications and server side testing of dynamic web applications. We begin with client side.

(Refer Slide Time: 03:56)



Client vs. server side testing

I

- For web applications, the user interface is on the **client** and the actual software is on the **server**.
- Clients and server are separated.
- Tester typically has no access to data, state or the source code on the server.
- We begin with client-side testing strategies.

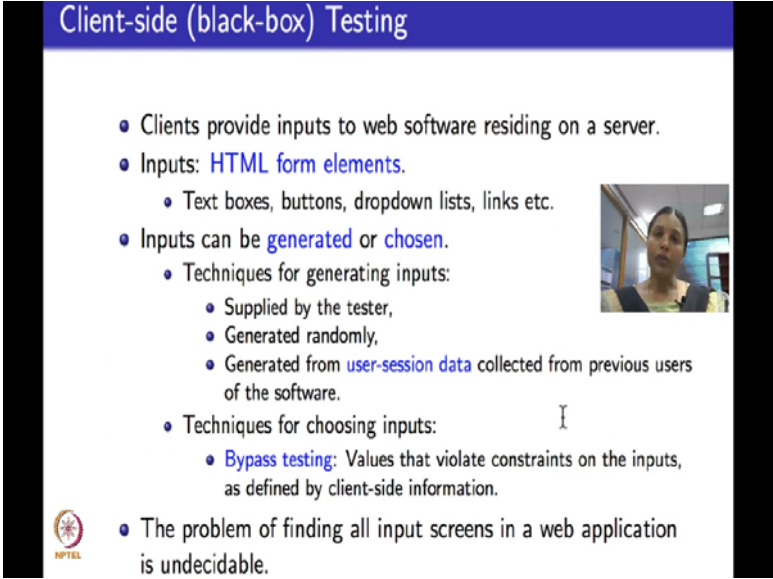
The slide features a blue header with the title 'Client vs. server side testing'. Below the title, there is a small 'I' icon. This is followed by a bulleted list with four items: 'For web applications, the user interface is on the client and the actual software is on the server.', 'Clients and server are separated.', 'Tester typically has no access to data, state or the source code on the server.', and 'We begin with client-side testing strategies.'. In the bottom left corner, there is a small circular logo with the text 'NPTEL' underneath it. In the bottom right corner, there is a small video inset showing a woman with dark hair, wearing a yellow top, speaking.

So, what is the difference what is client versus what is server? Server is the main place where the web application software resides different clients from different pages could access the server. Like for example, if you take our Indian railways ticket booking

system the main software will reside in a centrally located web server where the data base of the IRCTC will be made accessible. On that software various clients could access the web application based on each ones booking preferences. The clients are spread across India right. So, the user interface for web application is typically on the client and the actual software resides on the server. Clients and server are separated, geographically, the software, everything is separated.



Tester typically has no access to the data directly on the server that is a lot of data on the server, tester does not know this state of the server and typically black box tester for system level testing of web applications does not have access to the source code that is present in the server also. Now we going to see testing strategies that still manage to test in the presence of all these limitations. As I told you we will begin with client side testing in this lecture and then look at server side testing in the next lecture.

(Refer Slide Time: 05:21)



Client-side (black-box) Testing

- Clients provide inputs to web software residing on a server.
- Inputs: **HTML form elements**.
 - Text boxes, buttons, dropdown lists, links etc.
- Inputs can be **generated** or **chosen**.
 - Techniques for generating inputs:
 - Supplied by the tester,
 - Generated randomly,
 - Generated from **user-session data** collected from previous users of the software.
 - Techniques for choosing inputs:
 - **Bypass testing**: Values that violate constraints on the inputs, as defined by client-side information.
- The problem of finding all input screens in a web application is undecidable.

So, we what we going to do is system level black box testing of the client code. So, what does the client do? Client provides inputs to the web software. Where is the web software? The web software resides on the server.

What is a typical input that a client provides? The client could be filling up a form that is there is a web interface. So, the inputs could be HTML form elements. What are the various HTML form elements that are typically provided as a inputs to the client? They could be text boxes where you enter fields of texts. Like for example, if you go back to

the same ticket booking system typically there is a text box where you enter your user name followed by password that is encrypted and sent or they could be buttons, buttons in the sense of radio buttons, options for you to choose. Again going by the same ticket reservation example suppose you entered your beginning of the journey location and the destination location then the web system comes up with the list of trains that ply from the source to the destination.

And there is one button to choose which train you want. Based on the button that you press the corresponding train gets chosen and the availability and other details about that train is displayed to you. It could also be a drop down list a classical example of a drop down list is when you filling your address, you will realize that several websites give a drop down list of all the countries that are available. They will not let you type down your country. Similarly if you are filling your address in an Indian website there is a drop down list of all the states that are available.

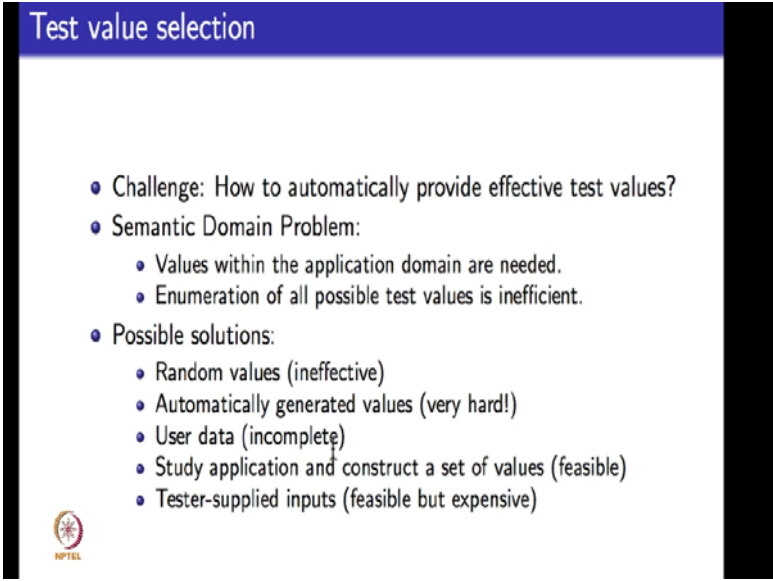
Those are various kinds of HTML form elements that we provide to create content for a HTML page. So, inputs to a client are basically HTML form elements of this kind. Now for testing these inputs can be chosen from a set or they could be generated from a set. So, what are the techniques that are used for generating inputs? Typically inputs are supplied by tester, tester could have lot of domain knowledge about the concerned applications and knows what to effectively test and supplies the test inputs or input could be randomly generated. Usually random generation of inputs is not considered very effective because you do not know what you are testing. Or, inputs could be generated from what is called a user session data which is basically collected from previous users of the software.

Each one has, each user has a software and then you store the users session data which is very easily storable if you have things like cookies and all that stuff and based on that you can manipulate that data to create test input. This is considered to be very effective and there are a couple of papers that discuss this technique of generating test inputs based on user session data and show how these techniques have been useful. I will point you to those papers towards the end of the lecture on web applications in the list of references. Now when it comes to techniques for choosing inputs to be given one popular technique that we will discuss in this lecture is what is called the technique of bypass testing.

What is the word bypass in English tell you? It tells you that you take some inputs, but you bypass, bypass in the sense you bypass by not giving inputs of right kind or not giving input values at all which means you give values that violate some constraints or the other on the inputs and the constraints where are they gotten from, they are gotten from clients side right. So, I will illustrate to you the technique of bypass testing through examples in this lecture. Now you might say now I told you inputs can be generated or chosen and we gave you high level overview of the various techniques.

But you might ask what about exhaustive testing, can I generate all possible inputs to a web application? And intuitive naturally correct answer to that would be the problem is very difficult. In fact, the problem is undecidable, you cannot find all inputs screens for a web application.

(Refer Slide Time: 09:24)



The slide is titled "Test value selection" in a blue header. It contains a bulleted list of points. At the bottom left of the slide content area is a small circular logo with the text "NPTEL" below it.

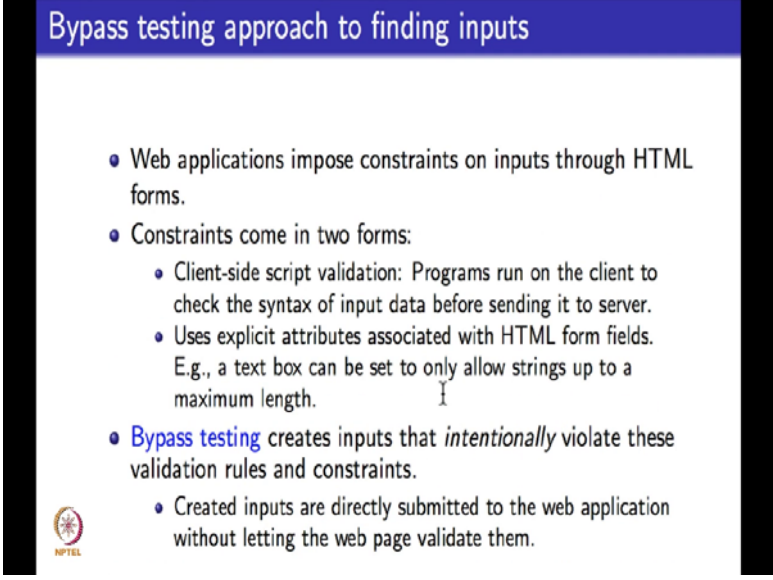
- Challenge: How to automatically provide effective test values?
- Semantic Domain Problem:
 - Values within the application domain are needed.
 - Enumeration of all possible test values is inefficient.
- Possible solutions:
 - Random values (ineffective)
 - Automatically generated values (very hard!)
 - User data (incomplete)
 - Study application and construct a set of values (feasible)
 - Tester-supplied inputs (feasible but expensive)

So, what we are going to do is we are going to do bypass testing before we move into bypass testing let us look at what is the test value selection problem. The challenge now is I want to be able to provide test values. How do I do? The value should be such that the values are within the application domain and I told you in the previous slide that I cannot generate all possible test values.

We discuss the random values could be possible, it is ineffective, automatically generate test values. How do you automatically generate test values? You intuitively understand that its hard problem right because this test inputs have a lot of structured. And if its user


session based data you may not you might get test values that are valid, but they might be in effective from the point of view of testing to identify of loss. We study an application and construct a set of values that is a feasible technique, bypass testing that we will be seeing comes under this class.

(Refer Slide Time: 10:16)



Bypass testing approach to finding inputs

- Web applications impose constraints on inputs through HTML forms.
- Constraints come in two forms:
 - Client-side script validation: Programs run on the client to check the syntax of input data before sending it to server.
 - Uses explicit attributes associated with HTML form fields. E.g., a text box can be set to only allow strings up to a maximum length.
- **Bypass testing** creates inputs that *intentionally* violate these validation rules and constraints.
 - Created inputs are directly submitted to the web application without letting the web page validate them.



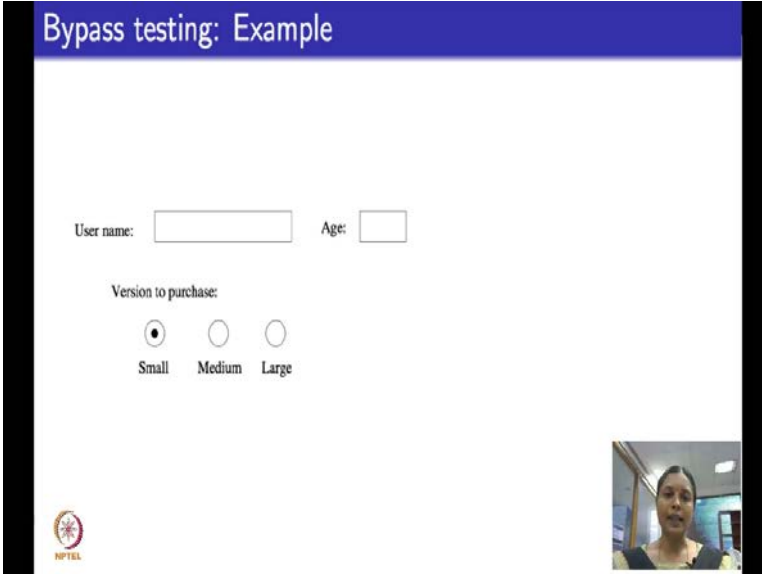
So, what is bypass testing? So, web applications typically impose constraints on the kind of inputs that they get through HTML forms right. What are the constraints that look like? Constraints can come in two different forms they could be constraint which is validated at the level of client itself by writing scripts for validating these constraints. What are these scripts? These scripts are basically programs software programs that run on the client to check whether the input data is of correct syntax like for example, let us say there is a field where you have to enter a name you obviously, cannot enter numbers there it has to be only alphabets.

Similarly another kind of constraints could be a explicit attributes that are associated with HTML form fields. Like for example, if you are entering age then you cannot enter a five digit number because this going to be nobody who is that old who is currently living. So, there could be constraints that could be validated by running scripts on the clients or they could be constraints that can be set by setting explicit attributes on the HTML form fields. We will see examples of both these kind of constraints. What does bypass testing do? Bypass testing now creates inputs that intentionally violate all these

constraints. So, they create inputs and submit the violated inputs directly to the server by bypassing the script validation that happens on the client side.

So, the inputs that violate the validation rules and constraints are directly submitted to the web application that resides on the server and the idea is to test and see how that web application reacts to these intentionally violated constraints. The client side script validation is bypassed hence the term bypass testing.

(Refer Slide Time: 12:07)



So, for an example let us say this is just a partial form of some web page where you ask to enter a user name, you are asked to enter your age and then you are asked to choose, you are buying a product let us say you are asked to choose the version that you want to purchase. It could be a small, it could be a medium or it could be a large.

This is how the normal input that is presented on a client page of a web application looks like. What are the classical inputs that you expect to be entered in this HTML form field? User name you expect a proper name to be entered which means what it should not have special characters it should not have numbers, age, you expect typically a two digit number to be entered you clearly do not want to enter an age which says its 700 or 1100 and so on right.

What will bypass testing do? If you enter wrong things typically at the client which is at the web page itself there will be simple validation rules like the kind that I showed you

here right. Script validation rules that will tell you that the data that you have entered is wrong correct it. Bypass testing, what it does is that it bypasses the clients side script validation, enters wrong data and tries to directly send that data to the server.

(Refer Slide Time: 13:25)

The screenshot shows a web form titled "Bypass testing: Example". The form has a header with a small video feed of a person. Below the header, there are two input fields: "User name:" and "Age:". The "User name:" field contains the text "Alan >Turing" and the "Age:" field contains the number "500". Below these fields, there is a section labeled "Version to purchase:" with three radio buttons: "Small", "Medium", and "Large". The "Small" radio button is selected. To the right of the form, there are two bullet points: "• User name should be plain text only." and "• Age should be between 18 and 110." In the bottom left corner, there is a logo for "NPTEL".

For example, if I am doing by pass testing for the same HTML form in the where I enter a user name I will insert a special character like this here I will say Alan greater than Turing. There should be a problem because special characters are not allowed in names, and age, I will enter some nonsensical number like 500 right.


Now, I will directly send this input by bypassing the client side validation to the software on the web server and see how the web software reacts. How do I directly send this input? How do I bypass the client side script validation? What do we do? We do the following.

(Refer Slide Time: 14:01)

Abbreviated HTML for example

```
<FORM>
  <INPUT Type="text" Name="username" Size=20>
  <INPUT Type="text" Name="age" Size=3 Maxlength=3>
  <P> Version to purchase:

  <INPUT Type="radio" Name="version" Value="150" Checked>
  <INPUT Type="radio" Name="version" Value="250">
  <INPUT Type="radio" Name="version" Value="500">
  <INPUT Type="submit" onClick="return checkInfo(this.form)">
  <INPUT Type="hidden" isLoggedln="no">
</FORM>
```

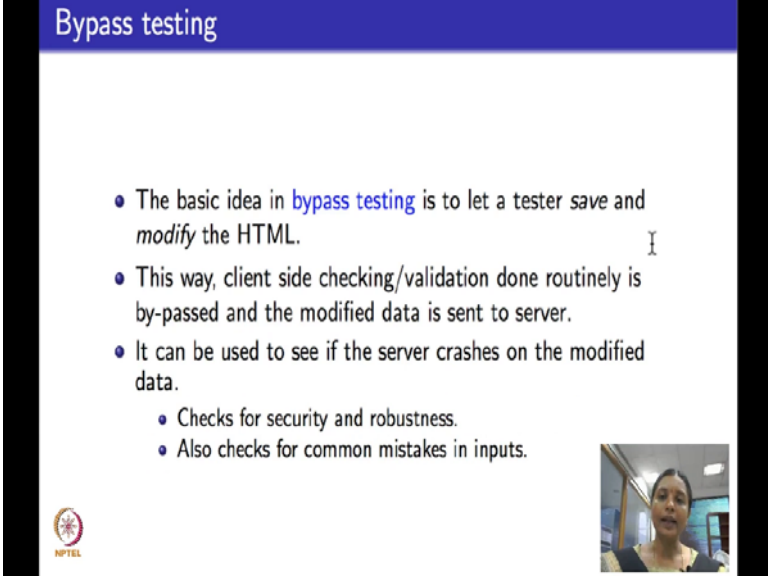


If the input was directly actually generated the corresponding HTML file that would be generated on the client site would look somewhat like this. So, there will be a form field, form tag in the HTML which will have so many inputs. It says the first input is of type text only text which means no special characters and numbers I used and the attribute that is entered will be user name which is passed to name and its size should be maximum 20 characters. You could increase or decrease this length based on what it is.

The next box is of type text the size should be the value that is entered is of age, size should be three and it cannot be more than length 3. Then the next box which says version to purchase could be its a radio button as we saw here these are call radio buttons small medium large, three versions to purchase small gets let us say some static value 150, medium gets another value 250, large gets an another value 500. And then typically there will be a submit button which I have not depicted here and then when you submit you on click right you return check info this form.

This is the place where the validation happens. Check info this form, check whether this form is correct. And then there is some other flag which is set which is logged in. I want to bypass this validation and directly if we wrong input to the server.

(Refer Slide Time: 15:30)



Bypass testing

- The basic idea in **bypass testing** is to let a tester *save and modify* the HTML.
- This way, client side checking/validation done routinely is by-passed and the modified data is sent to server.
- It can be used to see if the server crashes on the modified data.
 - Checks for security and robustness.
 - Also checks for common mistakes in inputs.

So, idea is to let a tester save this kind of a HTML and directly modify it. This way the client side checking or validation that is run routinely as a part of the client's script validation is bypassed and the modified data will be directly send to the server.


What can you use it from the point of view of testing? You can use it to see if the server crashes on the modified data and you can also use it for saying how the secure a server is secure how you robust it, does it a handle corrupt data well? It also checks for server better be robust for common mistakes on the inputs. So, it also checks if the server is robust enough for that. So, for the same example here, this is the HTML that corresponds to a normal form.

(Refer Slide Time: 16:20)

Bypassing Abbreviated HTML example

```
<FORM>
  <INPUT Type="text" Name="username" Size=20>
  <INPUT Type="text" Name="age" Size=3 Maxlength=3>
  <P> Version to purchase:

  <INPUT Type="radio" Name="version" Value="150" Checked>
  <INPUT Type="radio" Name="version" Value="250" >
  <INPUT Type="radio" Name="version" Value="500">
  <INPUT Type="submit" onClick="return checkInfo(this.form)" >
  <INPUT Type="hidden" isLoggedIn="no">
</FORM>
```





How do I bypass it I will modify this HTML, what are the modification that I will do to the HTML? I have depicted like this. I will remove this I will remove the part that says max length is three that let us me enter any garbage value that is a number for age.

And I will remove let us say some other modification, I could remove the value for large. And are the most important thing is I will remove this script corresponding to validation. This is the part where the script is validated, check info this form, I will remove it right. So, and now this HTML I will directly sent to the server. So, and then basically test how the server reacts to it.

(Refer Slide Time: 16:52)

Bypass testing: Client vs. server side

- Bypass testing modifies inputs.
- Can be done at the client side or server side.
- Client side inputs are *safer* and *easier* to handle.
- Server side inputs can be modified too, but, can be risky if they *corrupt* data in the server.




So, it modifies that the inputs it can be done on the clients side or server side I showed you how to do it on the client side through an example. Client side inputs are always say for an easier to handle. Server side, unless you are a web application developer of the software that directly resides on your server. We typically do not recommend that you do bypass testing on the server because they could end up corrupting data in the server right.

(Refer Slide Time: 17:20)

Types of client input validation

- Client side input validation is performed by HTML form controls, their attributes and client side scripts that access DOM.
- Validation **types** are categorized as **HTML** and **scripting**.
 - HTML supports syntactic validation.
 - Client scripting can perform both syntactic and semantic validation.

HTML	Scripting constraints
Length (max input characters)	Data Type (e.g. integer check)
Value (preset values)	Data Format (e.g. ZIP code format)
Transfer Mode (GET or POST)	Data Value (e.g. age value range)
Field Element (preset fields)	Inter-Value (e.g. credit # + expiry date)
Target URL (links with values)	Invalid Characters (e.g. <, &)



So, what are the typical types generically of the client input validation that happens? As I told you client side input validation is performed by HTML form controls, their

attributes, client sides scripts that act as the data. Validation types can be characterized as plain HTML form validations or validation for which you need to write extra program or script on the client. Plain HTML form validation are of the following type, you could write as a part of the form. I will go back to the example.

Like for example, I told you right if I enter a name I can give a restriction on its size I can tell you what are the characters that I expect. So, typical HTML validation is I can set a length which gives me the maximum length of the input characters I can set of fixed value which is preset to that value, I can describe what is the transfer mode is it a get or a post, I can describe what are the various preset field elements I can describe a particular target URL. These I can describe directly with HTML. What are the scripting constraints? I could check for the type of data it could be your number and integer it could be purely alphabets it could be alpha numeric but no special characters, or when its password should always have at least one special character these kind of things I can check.



I can check for format of data typically you are allow to export or upload only a pdf file, only if the file is zipped and so on. I can check the actual value of data which means suppose I say age should be between 18 and 100. I can actually check whether the value that is entered is within that range. I can check for example, inter values like for example, I can match the credit card number and also ask for expiry date like we do in banks, I can check for invalid characters.

So, all these validations plain HTML and for those that you write scripts are done on the clients side. Ech of these validations can be done and tested by using bypass testing.

(Refer Slide Time: 19:25)

Example client-side constraint rules

- Violate size restrictions on strings
- Introduce values not included in static choices
 - Radio boxes
 - Select (drop-down) lists
- Violate hard-coded values
- Use values that JavaScripts flag as errors
- Change "transfer mode" (get, post, ...)
- Change destination URLs





For example, what can I do? I can violate size restriction on this strings, I can include values that are not in static choices like radio boxes drop down list I can violate the hard coded values, I can use values that java scripts flags as errors, I change the transfer mode to gets to post, I can change the destination URL, I can do all these things to validate client side constraints to.

(Refer Slide Time: 19:48)

Example server-side constraint rules

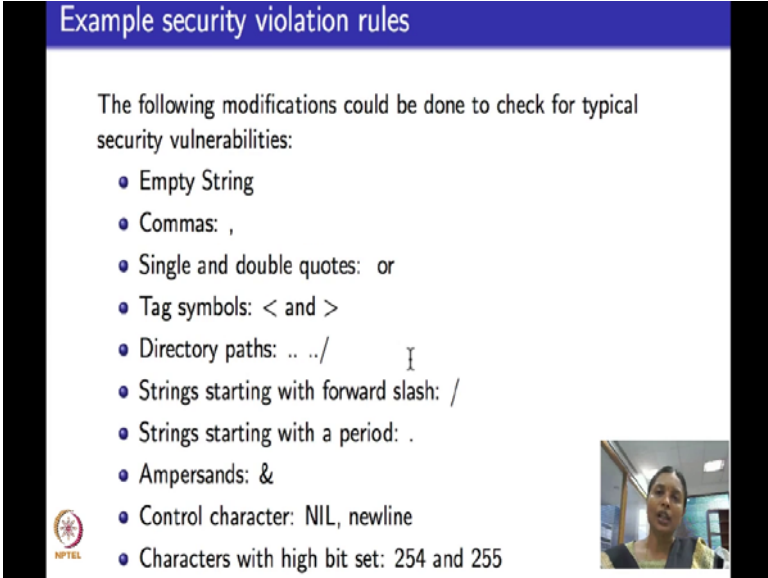
- Data type conversion
- Data format validation
- Inter-field constraint validation
- Inter-request data fields (cookies, hidden)



On the server side which I told you typically not recommended we can do bypass testing by doing data type conversion. We can change the data format of the data see how the server reacts to it.

We can change inter field constraints validation, like for example, you can say that if it is a visa card then it should be of this pattern right we can give master card with the visa pattern and see how the server reacts to it and so on. These are the various kinds of rules that can be check with bypass testing I can also check for various security violation rules.


(Refer Slide Time: 20:20)



The following modifications could be done to check for typical security vulnerabilities:

- Empty String
- Commas: ,
- Single and double quotes: ' or "
- Tag symbols: < and >
- Directory paths:/
- Strings starting with forward slash: /
- Strings starting with a period: .
- Ampersands: &
- Control character: NIL, newline
- Characters with high bit set: 254 and 255

HOPEL



I can give empty string, I can give a comma, I can insert special characters like single or double quotes, tag symbols, extra tag symbols that make this string syntactically invalid. I can change the path of the directory and so on. I can change the control characters by making new line paragraph and so on.

And typically all these will result in security violation in the server and if the server code or program is robust enough to handle these whatever you bypass and change server will correctly recognize it as invalid input and know how to handle it. So, this is an overview of bypass testing on the client side.

I will stop here for now. In the next lecture we will see how to do server side testing of web applications.


Thank you.

Software testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 47
Testing of Web Applications and Web Services

Hello we are in the middle of week 10. I was doing web applications testing.

(Refer Slide Time: 00:17)



The slide is titled "Overview" and contains a bulleted list of topics. The list includes: Introduction to relevant aspects of web applications, Issues in testing of web applications, Testing static hyper text web sites, and Testing dynamic web applications. Under the last item, there are two sub-bullets: Client-side testing of web applications and Server-side testing of web applications. The NPTEL logo is visible in the bottom left corner, and a small video feed of the professor is in the bottom right corner.

- Introduction to relevant aspects of web applications.
- Issues in testing of web applications.
- Testing static hyper text web sites.
- Testing dynamic web applications.
 - Client-side testing of web applications.
 - Server-side testing of web applications.

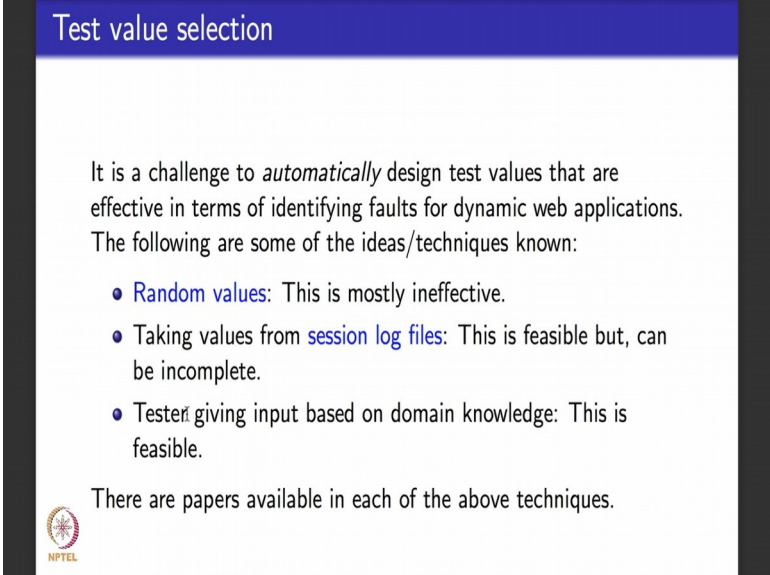
And we had broadly categorized web applications testing into 3 categories. Testing, static, hyper text web sites which I did as a part of my first lecture where we also discussed issues of testing web applications. Last class I taught you about client side testing of web applications in particular one technique called bypass testing. What I want to focus on in this lecture is server side testing of web applications. We will try to derive 2 specific graph models that correspond to server side testing, and I will tell you what those represent as far as testing the web applications software that present on the server that is concerned.

After this we will move on and do testing of object oriented applications.

So, I will skip past this slides that we have already discussed in the context of testing web applications. When we did dynamic testing I told you 2 categories, client side and server side. This is what you saw last time, client side testing, which basically involved

bypass testing is the technique that we learnt. I will move on to server side testing, this is where we had stopped last time.

(Refer Slide Time: 01:25)




Test value selection

It is a challenge to *automatically* design test values that are effective in terms of identifying faults for dynamic web applications. The following are some of the ideas/techniques known:

- **Random values:** This is mostly ineffective.
- Taking values from **session log files:** This is feasible but, can be incomplete.
- **Tester giving input based on domain knowledge:** This is feasible.

There are papers available in each of the above techniques.



We talked about how to select test cases when we do server, server side testing. Just to quickly recap we are doing system level testing of web applications. When I do a method level testing or I do a component level testing then whatever we discussed in the traditional ways like, graph based logic based, mutation testing they all can be applied.

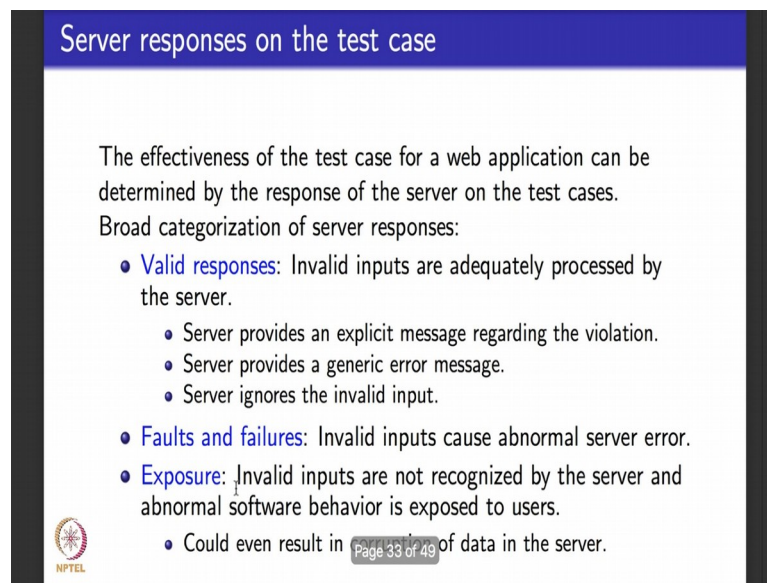
I am assuming that all that is done now. I have the full code integrated into the server with the clients available and I am doing system level testing which comes towards the end of testing phase. And I want to focus on system level testing of web applications on the server side. When it comes to that, when we say test values on the server side, what is the big challenge? The idea is to automatically design test cases or test values that are effective in terms of finding faults. For what kind of web applications, for dynamic web applications, that is what we are concentrating on.

This is just recapping what we discussed. One option is to select random test cases and test. This is useful sometimes, but not specifically effective all the time. The other idea is to take values from session log files. I will not be discussing this technique in detail, but I will point you to a reference material at the end of this lecture, where you can get some more information about how to test based on user session data. The third option is to get

for the tester to give inputs based on domain knowledge. One technique that we already discussed in this category was bypass testing, which we did on the client side.

One more technique that we will discuss for the server side based on this, is what we are going to discuss today which will involve deriving graph models from the server side. And as I told you I will give you pointers to papers where you can know in detail about all these techniques.


(Refer Slide Time: 03:20)




Server responses on the test case

The effectiveness of the test case for a web application can be determined by the response of the server on the test cases.

Broad categorization of server responses:

- **Valid responses:** Invalid inputs are adequately processed by the server.
 - Server provides an explicit message regarding the violation.
 - Server provides a generic error message.
 - Server ignores the invalid input.
- **Faults and failures:** Invalid inputs cause abnormal server error.
- **Exposure:** Invalid inputs are not recognized by the server and abnormal software behavior is exposed to users.
 - Could even result in  of data in the server.



Before we move on and discuss server side testing, when we say I want to define test cases to do system level testing that will pull out errors, what could be errors related to server side testing of web application? So, errors could be handled by the server, they could be detected and handled by the server, they could be detected and not handled by the server in the sense that they cause an error.

In addition to that, not only causing an error it could result in permanent damage to the server in the sense of corrupting the data on the data base that is present in the server and so on. So, server when it handles the test case the error caused by the effective test case, server could produce a valid response. What is a valid response? A valid response is any response that is produced by the server in response to handling an invalid input that is generated by a test case. Servers aware of the fact that the input is invalid and it responds in one of the following way. It responds by providing an explicit message saying that

there is a problem with this input, this input is not of this format and it explains the violation.

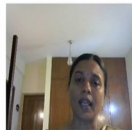

It could respond by providing a generic error message which is not fine tune to the kind of violation that happens or server could just ignore the invalid input. What are we mean by ignoring the invalid input? It recognizes the fact that it is an invalid input, ignores in the sense of does not intimate the tester or the client that the input is invalid, ignores it goes ahead and does its job normally. The second categorization of server responses could result in a fault or a failure. A fault is an internal fault, it may or may not propagate to being visible in the output. A failure propagates to being visible in the output. Both the cases there is a server error and it is abnormal in nature.

And when the failure gets exposed what happens is that the invalid inputs that are not recognized by the server, the user also gets to know that it is not recognized by the server, because there is a visible exposure of that error from the information that the server provides to the user. Or the server may not provide any information to the user. In fact, a very worse situation can happen which is the invalid input is so bad that the server does not only produce abnormal errors, it results in corruption of data in this server like I told you it could corrupt the data present in the data base in the server fix sometimes could cause more damage than intended.

(Refer Slide Time: 06:02)

Server-side (white-box) testing

- If server-side source code is available, we can use our usual graph models to test the server.
- Control flow graph exhibits only *static* models, not effective for web applications.
- **Presentation layer** of a web application contains the software and is useful to do testing.
- For software in the presentation layer, two graph models exist:
 - **Component Interaction Model (CIM)**
 - **Application Transition Graph (ATG)**



So, when it comes to server side white box testing what are we going to do? If the source code of the server is available as I told you to do method level testing, unit level testing, we can use our usual graph models to test the server. But the problem is for system level testing of server, graph models like control flow graphs, do not really tell you about the dynamic behavior of the server. They are static models that describe the structure of the code that is return for the server, they do not talk about how this server put together with the client behaves as a system that is put together.

So, what we want to do is we want to derive and work with 2 new graph models that talk about the system level behavior from the server side. When we talk about the server side code you remember 2 classes ago, I presented to you the architecture of web application which involves several different layers, there was a layer containing the data then there was a layer that contains, it is user interface. In between these 2 layers there was a presentation layer. What we are focusing on is the code that is present that corresponds to the server as it is given in the presentation layer. From this code that is present in the presentation layer of architecture of a web application we are going to derive 2 different graph models.

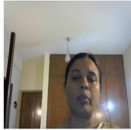

The first graph model that we are going to work with is called component interaction model, abbreviated as CIM. The second graph model that we are going to work with is called application transition graph model abbreviated as ATG. I will first begin with the CIM model then move on to ATG model.

(Refer Slide Time: 07:43)

Atomic sections

We need the notion of **atomic sections** to understand the graph models.

- An **atomic section** is a section of HTML with the property that if any part of the section is sent to a client, the entire section is.
 - May include JavaScript
 - All or nothing property
- A HTML file is an atomic section.
- A **content variable** is a program variable that provides data to an atomic section.
- Atomic sections may be empty.




So as I told you please remember we looking at the server side code as it is present in the presentation layer of a web application that is dynamic in nature. So, when it is dynamic in nature, a particular website can be created in different ways in respond to different requests.

And it is created by dynamically generating HTML forms and when HTML forms are dynamically generated a part which generated in sections of fragments. And there are sections of fragments that are specifically called atomic sections that are very important for us. So, what is an atomic section? An atomic section is a section of a HTML file with the following property, if one part of the section is sent to the client then the entire part of the section is sent. Before moving on I will take an example and explain to you what it is.

(Refer Slide Time: 08:32)

Atomic sections: Example	
	PrintWriter out = response.getWriter();
P1 =	out.println("<HTML>") out.println("<HEAD><TITLE>" + title + "</TITLE></HEAD>") out.println("<BODY>")
	if(User) {
P2 =	out.println("<CENTER> Welcome!</CENTER>"); for (int i=0; i<myVector.size(); i++) if (myVector.elementAt(i).size > 10)
P3 =	out.println("<p>" + myVector.elementAt(i) + "</p>"); else out.println("<p>" + myVector.elementAt(i) + "</p>");
P4 =	} else
P5 =	{ }
P6 =	out.println("</BODY></HTML>"); out.close ();

 Atomic sections are coloured in red. P1, P2, P3 are empty atomic section. Content variables are coloured green.

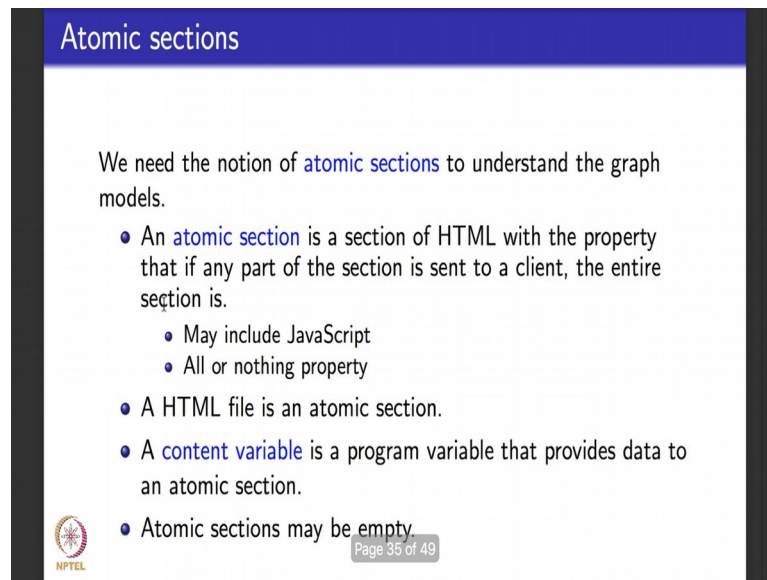
Page 36 of 49

So, here is how the code looks like, the this thing that I bracketed as different rows in the table corresponds to some server side code present in the presentation layer.

So, it first gets a response from the client, and then it prints out this chunk. First remember it is to print out a HTML file to be rendered in the web browser interface of the client. And every HTML file begins with the HTML tag followed by a header tag followed by a body tag. So, I have put all these 3 print statement that print the begin of the HTML header tag, the HTML main tag followed by the header tag, which prints the title and ends the head tag followed by the beginning of the body tag. All these 3 are printed together.

So, they are put as one row in the table and constitute one atomic section. After that if the user is authenticated. If the user is a valid user then a welcome message is printed and input is being read. If the user is an invalid user go down here which is the else part then nothing is printed, and the HTML file is ended and closed. So, an atomic section is a chunk of HTML file such that if one part of the file is output to the client, is sent for printing by the client at the end, then the entire part is. Here this is one chunk that is printed this is the next chunk that reads the, if the users validated prints the message, this is the next chunk that is printed. So, this is the next chunk which is empty.

(Refer Slide Time: 10:14)



The slide has a blue header with the title "Atomic sections". The main content is on a white background with a black border. It starts with a paragraph: "We need the notion of atomic sections to understand the graph models." This is followed by a bulleted list: "• An atomic section is a section of HTML with the property that if any part of the section is sent to a client, the entire section is." (indented), "• May include JavaScript" (indented), "• All or nothing property" (indented), "• A HTML file is an atomic section.", "• A content variable is a program variable that provides data to an atomic section.", and "• Atomic sections may be empty." (indented). In the bottom left corner is the NPTEL logo. In the bottom right corner, there is a small grey box containing the text "Page 35 of 49".

Atomic sections

We need the notion of atomic sections to understand the graph models.

- An atomic section is a section of HTML with the property that if any part of the section is sent to a client, the entire section is.
 - May include JavaScript
 - All or nothing property
- A HTML file is an atomic section.
- A content variable is a program variable that provides data to an atomic section.
- Atomic sections may be empty.

NPTEL

Page 35 of 49

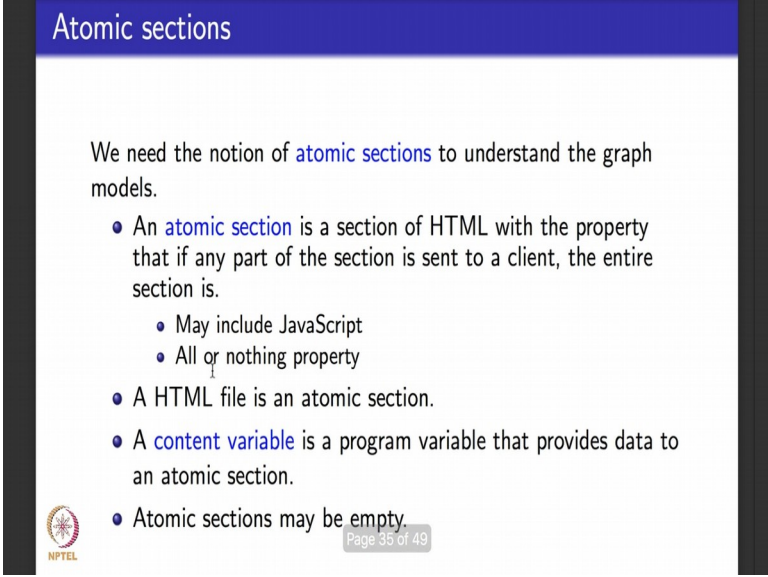
So, that is what is described here and atomic section is a section of the HTML file with the property that if any part of the section of the HTML file is sent to the client the entire section is. Atomic sections might themselves include scripts return in Java script. It could be empty, atomic sections basically satisfy all or nothing. If the thing is printed it is printed in its entirety or nothing is printed. One full HTML file could be printed as an atomic section. Atomic sections could be empty. Another related terminology that we will need is the notion of a content variable. What is the content variable? It is a program variable that provides data to an atomic section.

So now if you go in this example these things that I have bifurcated by using these lines to see how the code is fragmented, the ones that are marked in red on this column on the left hand side represent atomic sections. So, there are 6 different atomic sections. If you notice there is one atomic section P5 which is completely empty. In this scenario, why is it empty? It is empty because the user is not been validated. So, they do not want to print the welcome message and read the data and print it. So, they print nothing, but they close the body and HTML file so that the empty HTML file can be rendered back on the client's web browser. Content variables as we see are colored in green.

So, here title is a content variable. What is a content variable? That generates data dynamically. So, title of the web page is dynamically generated. For example, let us say if bank page it could print welcome message followed by your name as a part of the title.

And the name part of the welcome message is dynamically generated. This data, whatever it is which is stored in this vector element, it could be data related to your statement if it is a bank page it could be data related to your marks, if it is your marks page it could be anything this data that is dynamically generated is also called content.

(Refer Slide Time: 12:17)



Atomic sections

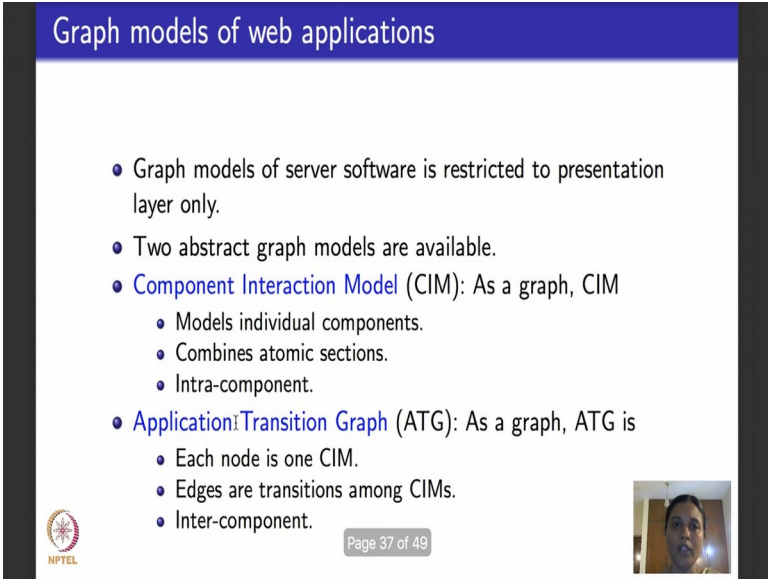
We need the notion of **atomic sections** to understand the graph models.

- An **atomic section** is a section of HTML with the property that if any part of the section is sent to a client, the entire section is.
 - May include JavaScript
 - All or nothing property
- A HTML file is an atomic section.
- A **content variable** is a program variable that provides data to an atomic section.
- Atomic sections may be empty.

NPTEL Page 35 of 49

So, I hope these 2 terminologies are clear, atomic section and content variable.

(Refer Slide Time: 12:21)



Graph models of web applications

- Graph models of server software is restricted to presentation layer only.
- Two abstract graph models are available.
- **Component Interaction Model (CIM)**: As a graph, CIM
 - Models individual components.
 - Combines atomic sections.
 - Intra-component.
- **Application Transition Graph (ATG)**: As a graph, ATG is
 - Each node is one CIM.
 - Edges are transitions among CIMs.
 - Inter-component.

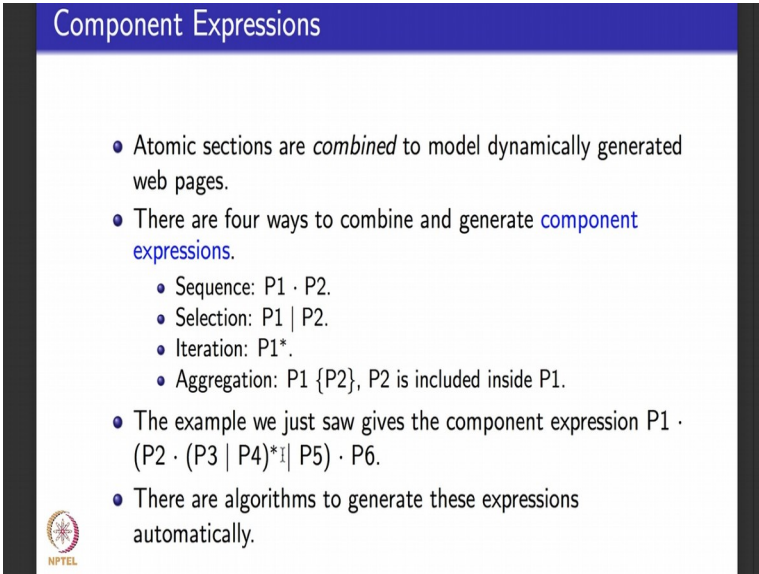
NPTEL Page 37 of 49

Moving on, our 2 graph models that we will be considering will be built on atomic sections. And then further on the first level graph model. So, component interaction

models CIM as a graph, what does it do? Models individual components combines atomic sections. It is intra component means it talks about interaction within a component. Nodes of a component interaction model graph will be atomic sections; transitions will tell you how to move from one section to the other, with that thing it will model one particular component of a web server software that is present in the presentation layer.


Building on the CIM model, we will later on define the application transition graph or the ATG model where each node of the ATG model will be one CIM graph and edges would be transitions amongst the CIM graph. Because each node is one CIM graph and application transition graph is a inter component graph, that is it models interactions across components. I will first begin with CIM.

(Refer Slide Time: 13:30)



Component Expressions

- Atomic sections are *combined* to model dynamically generated web pages.
- There are four ways to combine and generate **component expressions**.
 - Sequence: $P1 \cdot P2$.
 - Selection: $P1 \mid P2$.
 - Iteration: $P1^*$.
 - Aggregation: $P1 \{P2\}$, P2 is included inside P1.
- The example we just saw gives the component expression $P1 \cdot (P2 \cdot (P3 \mid P4)^* \mid P5) \cdot P6$.
- There are algorithms to generate these expressions automatically.

 NPTEL

So, in before we understand CIM we need to know what component expressions are. Atomic sections which we saw little while ago, they can be combined to model dynamically generated web pages. How will you combine atomic sections? If you remember when we did mutation testing, I had introduced you to regular expressions. Regular expressions is one standard way of combining atomic sections.

So, this sequential operator, this is P1 is one atomic section. P2 is one atomic section. $P1.P2$ means atomic section P1 is followed by atomic section P2 in a sequential way in this order. This is a same syntax as that of sequential concatenation in a regular

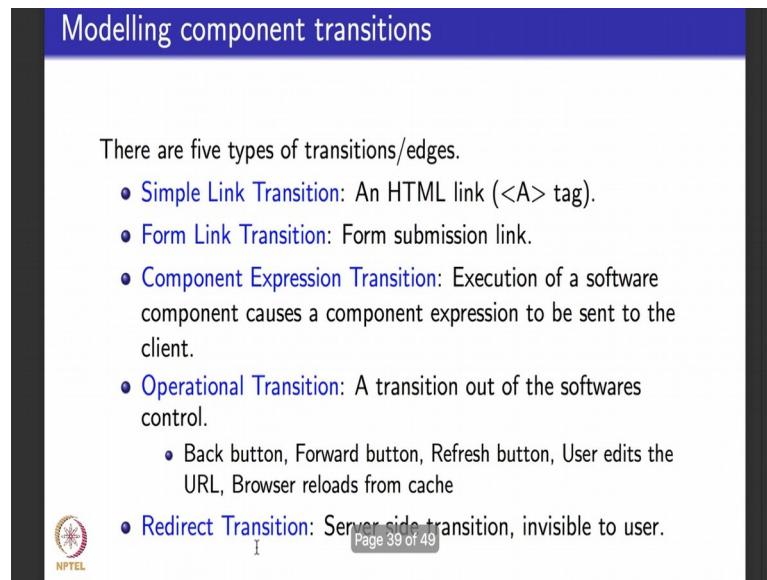
expression. Selection which corresponded to + as we defined it in regular expression is union which means either print P1 or print P2, print one of P1 or P2, iteration is again very similar to the regular expression iteration print 1, 0 or more occurrences of the atomic section P1. In addition to these three standard regular expression like operators we also have the operator of aggregator.

What is aggregator do? If the syntax is like this: it is says P1 and then you open these flower braces or curly braces and then you print P2. So, the way to read it is first P1 is printed as an atomic section. And inside the atomic section P1, P2 is printed right. So, these are the 4 ways to combine component expressions. If you go back to this example that we saw, atomic section P1 is printed once, this begins the HTML file, and then this is a decision statement which corresponds to the internal code present in the server. If this decision is positive, turns out to be true, which means the user is validated then atomic section P2 is printed.

Where P2 begins with the welcome message, then what does this one do? This one goes through the input the data containing, the input is present in this array called my vector. So, this is a for loop that goes through that, and as it is going through the thing in different paragraph, in a new line break it prints each element as it is reading from the input, and it keeps doing that and when it ends it exits and comes out. So, this is printed as many times as there are inputs to be read. So, the sequential composition of that thing would be you print P1 and then you do P2 and then you either print P3 or P4 as many times as the for loop would run or if the user is not validated print the empty atomic section P5.

Whatever you do, follow it with P6 which closes the HTML body and main HTML type. So, this is how atomic section are combined, and this is the graph also corresponding to the component interaction model. Given a particular web server code there are algorithms to generate these expressions automatically as I told you I will give you references from where you can get these algorithms to do it.

(Refer Slide Time: 16:48)



Modelling component transitions

There are five types of transitions/edges.

- **Simple Link Transition:** An HTML link (<A> tag).
- **Form Link Transition:** Form submission link.
- **Component Expression Transition:** Execution of a software component causes a component expression to be sent to the client.
- **Operational Transition:** A transition out of the softwares control.
 - Back button, Forward button, Refresh button, User edits the URL, Browser reloads from cache
- **Redirect Transition:** Server side transition, invisible to user.

NPTEL Page 39 of 49

So now, what are the various kinds of transitions that can occur in a component interaction model? There are 5 different kinds of transitions, a transition could be the result of a simple HTML link.

They anchor tag which many of you might know is one such example, I just in a part of a HTML text, I just use the anchor tag and hyper link to another page. So, this is an example of a static link or a simple link. The next example of a transition could be a form link transition, which is basically you typing some data like your user ID and password and you are exposed to a form from which you type in this data and you submit. When you submit you create what is called a form link. The third kind of link could be a component expression transition which is particular execution of a particular software component causes one whole expression to be sent to the client.

An expression like this like the one that we saw earlier. The fourth could be an operational transition, what is this mean? This means that is the sudden transition that comes as an a unplanned input from the user. Typically if you are accessing your bank side or if you are doing your online shopping for some reason and you have typed in your credentials and you are accessing your bank's site, you will often find the information which says do not press back or refresh button, why? Because the particular data is being encrypted and sent across from the client which is a instance of your

machine from which you are accessing your bank to the server, this takes a while for a server to process that data validate it and print it.


And it might if you run out of patients and press back button then you interrupt to the processing of the server. And those kind of things are called operational transitions. It is a transitions that was not as a part of the planned routine that disrupts the operations that the server perform some particular client input data. Typically by pressing back button, forward button refresh button or you change the URL browser reloads from cache, all these things cause abnormal operational transitions. The other thing could be what is called redirect transition which is an internal transition done by the server.

It is typically not visible to a user who sitting on the client. To give you an example of what are redirect transition could be like for example, the server code could be modularized and for the server to be able to particularly, let us say it has to need, it has to fetch some kind of data from a data base which is sitting in another server, and let us say the main web server has access to the data base server this main web server could do a redirect transition to the data base server, get the data or validate the data and accordingly respond. So, that kind of transitions have called redirect transitions. So, transitions that are simple hyper links, transition that of forms, transitions that corresponds to component expressions, transitions that are abnormal which cause abnormal interruption in server processing of data and these are transitions that are internal to the server.

(Refer Slide Time: 19:58)

CIM Example: gradeServlet

	<pre>ID = request.getParameter ("Id"); passWord = request.getParameter ("Password"); retry = request.getParameter ("Retry"); PrintWriter out = response.getWriter();</pre>
P1 =	<pre>out.println ("<HTML><HEAD><TITLE>" +title+ "</TITLE></HEAD><BODY>") if ((Validate (ID, passWord)) {</pre>
P2 =	<pre>out.println (" Grade Report "); for (int l=0; l < numberOfCourse; l++)</pre>
P3 =	<pre>out.println("<p>" +courseName(l)+ "" + courseGrade(l)+ "</p>"); } else if (retry < 3) {</pre>
P4 =	<pre>retry++; out.println("Wrong ID or wrong password"); out.println("<FORM Method='get' Action='gradeServlet'>"); out.println("<INPUT Type='text' Name='Id' Size=10>"); out.println("<INPUT Type='password' Name='Password' Width=20>"); out.println("<INPUT Type='hidden' Name='Retry' Value='"+(retry)+">"); out.println("<INPUT Type='submit' Name='Submit' Value='submit'>"); out.println("Send mail to the professor<A>");</pre>



So, how does a CIM model look like? We will expand on what we did in this model here. I just told you here for my vector element. Let us say this is something that prints the marks of a particular student. A student is waiting for results let us say of his tenth standard or twelfth standard or GATE exam. Results are going to be made available online on a particular date and if the user is valid, this actually reads the data of the score or the marks corresponding to the subject and prints it out.

So, what I will do is I will retain some parts of it the later parts of this, but I will elaborate on this part of it. So, this is an example of a GradeServlet, grade in the sense of a student grades. So, you start by saying you request for ID request for password. If the password is wrong then you request for retry and then you get a response. And then you start printing, this is the header tag, this is the body tag. If the ID and password are correct then you start printing the grade report. This is the title and let us say some student might have written 5 courses, 6 courses, 4 courses, different number of courses. So, you run a loop through the number of courses print the course name and the grade.


If the password is wrong then you are allow 3 attempts to correct your password. So, in the retry is the variable that captures the number of attempts. If retry is still less than 3 then you say your password or ID was wrong. And again redo this thing of password retries, submit and update retry here. And when the number of attempts cross then you say I have sent a mail to professor indicating that this is a problem. So, is it clear? And

then towards after this P4, I ended with this same P5 and P6 which I have not repeated which is empty, P6 closes the HTML tag. So, let us say this is an example of a code on the server which prints the grades.

(Refer Slide Time: 21:53)

CIM for gradeServlet

- Start page S is login.html.
- $A = \{P1, P2, P3, P4, P5, P6\}$.
- $CE = \text{gradeServlet} = P1 \cdot ((P2 \cdot P3^*) \mid P4 \mid P5) \cdot P6$.
- Transitions $T = \{\text{login.html} \rightarrow \text{gradeServlet}[\text{get}, (\text{Id}, \text{Password}, \text{Retry})], \text{gradeServlet.P4} \rightarrow \text{sendMail}[\text{get}, ()], \text{gradeServlet.P4} \rightarrow \text{gradeServlet}[\text{get}, (\text{Retry})]\}$

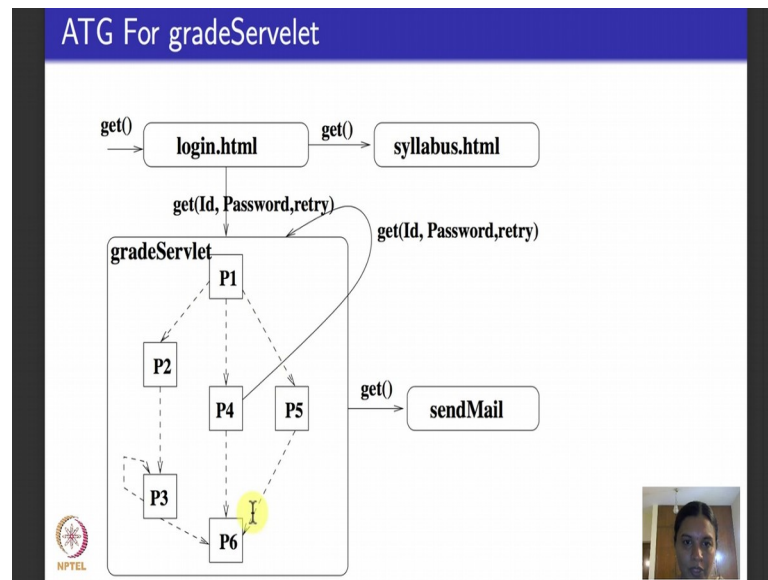


Page 41 of 49

I

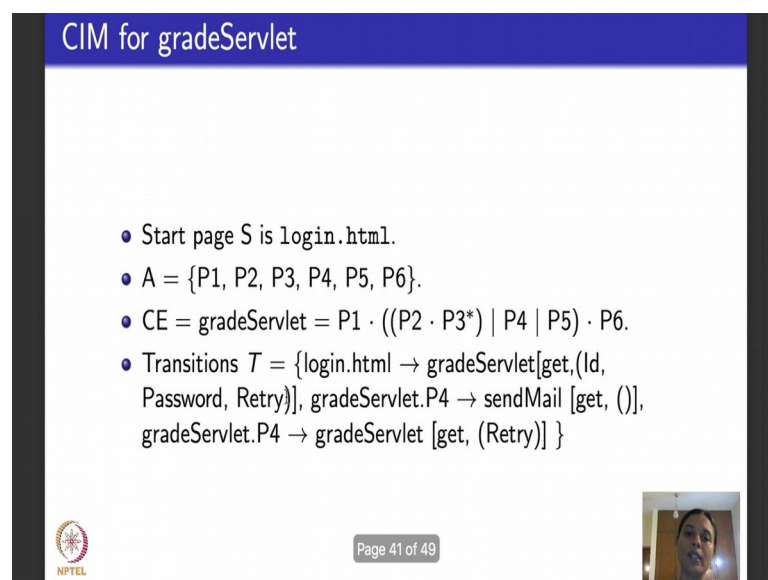
So now let us generate the component interaction graph model for this. Start page will be login dot HTML which is not described here. And then atomic sections will be P1, P2, P3, P4 which are these, P5 and P6 generated from the earlier side and like we did last time always do P1 which is the header information. And then do P2 which is validating and print P3 as long as there is code, our P4 which is retry P5 retries crossed, ended with P6 which is the final ending of HTML tag. So, what are the transitions from login dot HTML I go into the grade servlet which is this component this entire component and inside grade servlet I can do transitions from P4 which talks about sending mail or we talk about number of retries of a password and so on. This is that part.

(Refer Slide Time: 22:50)



Just concentrate only on this where I am pointing my mouse to, P1 could be followed by P2 any attempts of P3 or P6 or P1 could be followed by P4 followed by P6 which corresponds to password retrials up to 3 times, or P1 could be followed by P5 could be followed by P6 which corresponds to failure after the 3 retrials and empty page is printed.

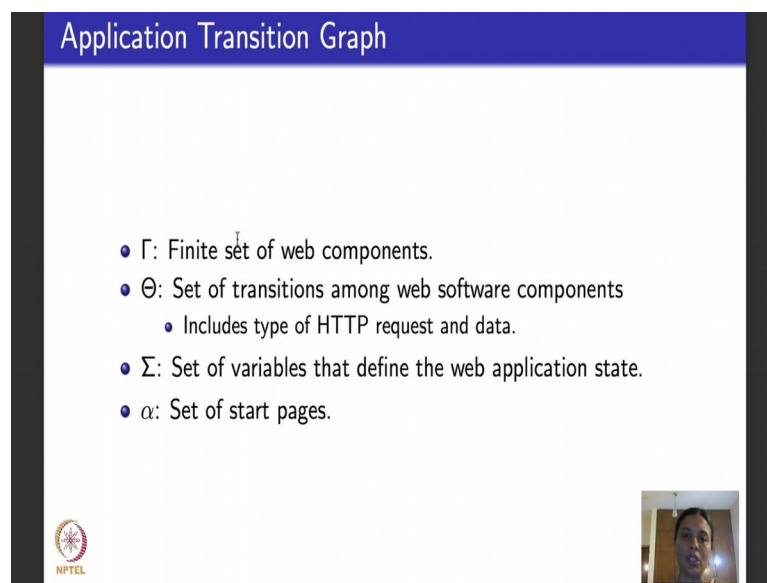
(Refer Slide Time: 23:16)



So, this is how a CIM looks like. I hope this example explain to you to a certain extent how a CIM model looks like.

Now, moving on we will look at how an application transition graph model looks like. If you go back to the slide where I introduced these 2 models to you, I said an application transition graph is also a graph where each vertex or a node is one CIM graph and edges are transitions amongst the components. So, it models inter component behavior at the presentation layer. Let us say to define an application transition graph, let us say we have a finite set of web components called Γ , Θ is the set of transitions amongst the web software components.

(Refer Slide Time: 23:50)



The slide is titled "Application Transition Graph" in a blue header. It contains a bulleted list of symbols and their meanings. In the bottom left corner, there is an NPTEL logo. In the bottom right corner, there is a small video inset showing a person's face.

- Γ : Finite set of web components.
- Θ : Set of transitions among web software components
 - Includes type of HTTP request and data.
- Σ : Set of variables that define the web application state.
- α : Set of start pages.

This include this could include HTTP requests sending of data, abnormal button presses any kind of transitions that we discussed earlier. And Σ is the set of content variables that defines this state of a web application and α is the set of start pages.

(Refer Slide Time: 24:17)

ATG for gradeServlet

- $\Gamma = \{\text{login.html}, \text{gradeServlet}, \text{sendMail}, \text{syllabus.html}\}.$
- $\Theta = \{\text{login.html} \rightarrow \text{syllabus.html}[\text{get},()], \text{login.html} \rightarrow \text{gradeServlet} [\text{get},(\text{Id},\text{Password},\text{Retry})], \text{gradeServlet.P4} \rightarrow \text{sendMail}[\text{get},()], \text{gradeServlet.P4} \rightarrow \text{gradeServlet} [\text{get},(\text{Retry})]\}$
- $\Sigma = \{\text{Id}, \text{Password}, \text{Retry}\}.$
- $\alpha = \{\text{login.html}\}.$

NPTEL

Page 43 of 49

So in the grade servlet example they could be 4 different web files login dot HTML whose code I didn't give you let us say it is a static website that just has a form, grade servlet whose code we saw send mail whose code I didn't give you, but let us say it is similar to the grade servlet in the sense that it fixes up the email ID from the data base and sends an email and then there is another static page let us say syllabus dot HTML.

So, the transition could be from login page you could go to a page where you get the syllabus dot HTML, fetch your data or from the login page you could directly go to a page where you print the grade details provided ID password are valid and retries within the allowed amount or, and from the grade servlet page you could go to a page where the grade report is sent by email or, from the grade servlet page you can go back to the grade servlet page because one of the attempts to enter user ID and password failed and you are still retrying within the 3 attempts to get it right.

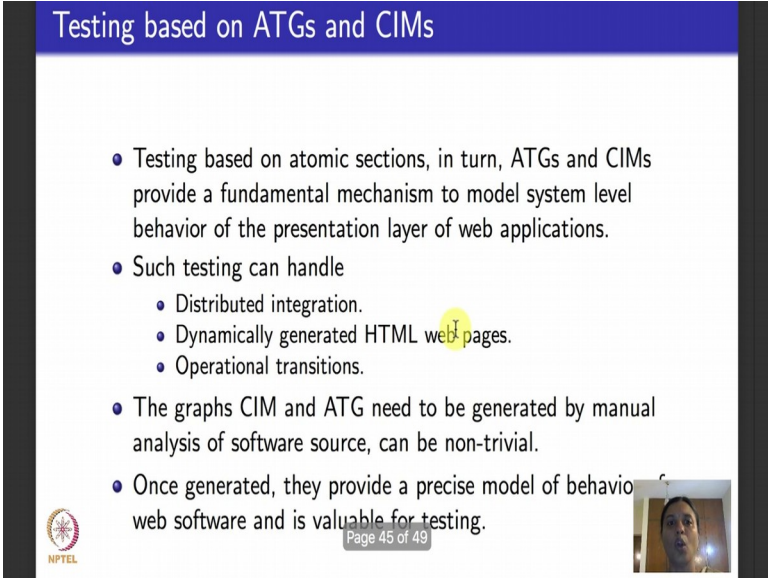
And what is this set of action set of actions or the variables which is defined are what is the ID sorry, there should be ID not IS, ID, ID name password and the number of retries the begin page is the login dot HTML page. So, here is how the ATG graph looks like. I go I through a get request I go to the first page which is a login page, and then I can go to the syllabus page where I printed somebody might just want to know this syllabus you print it and you stop nothing more to do. From the login page if I get ID and password I

am allowed 3 retries as long as $\text{retry} < 3$. So, here is where I print the beginning thing if my ID and password is wrong, I go back and retry, this is presented lose ended here.

But the internal code server will ensure that this transition happens only 3 times. But if everything is correct then the grade card and the marks are printed here and you exit. The number of retries exceed 3, another ID and password still do not turn out to be right, then, you print the empty page and exit. Whatever it is you go to the send mail page. So, this whole entity describes the complete behavior of the server side software, as it is present in the presentation layer and inside this ATG, is one of the pages gradeservlet and one of the components gradeservlet.

We actually sort of zoomed in and looked at the CIM model which I discussed to you. Similarly you could have another CIM model here inside send mail, because this could also be a dynamically generated server code. But these this is probably static because it is just prints the syllabus, and this could probably have a form. So, if you kind of zoom in to each of these nodes, they could be similar internal graphs that correspond to the CIM graphs for each of this modules. So, this is how you generate graphs corresponding to system level dynamic behavior of a web software that is present in it is presentation layer.

(Refer Slide Time: 27:20)



The slide is titled "Testing based on ATGs and CIMs" in a blue header. It contains a bulleted list of four points. The first point states that testing based on atomic sections, in turn, ATGs and CIMs provide a fundamental mechanism to model system level behavior of the presentation layer of web applications. The second point states that such testing can handle distributed integration, dynamically generated HTML web pages, and operational transitions. The third point states that the graphs CIM and ATG need to be generated by manual analysis of software source, can be non-trivial. The fourth point states that once generated, they provide a precise model of behavior of web software and is valuable for testing. In the bottom left corner, there is an NPTEL logo. In the bottom right corner, there is a small video inset showing a person speaking. A small text box at the bottom center of the slide reads "Page 45 of 49".

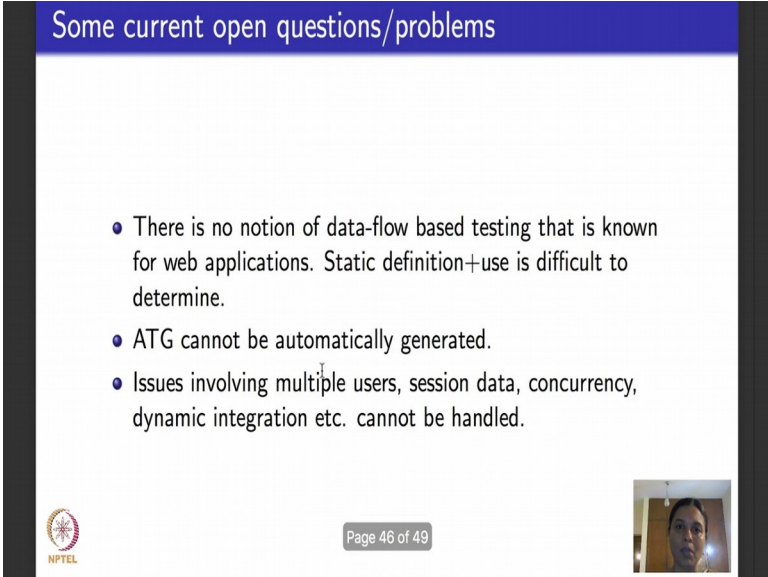
- Testing based on atomic sections, in turn, ATGs and CIMs provide a fundamental mechanism to model system level behavior of the presentation layer of web applications.
- Such testing can handle
 - Distributed integration.
 - Dynamically generated HTML web pages.
 - Operational transitions.
- The graphs CIM and ATG need to be generated by manual analysis of software source, can be non-trivial.
- Once generated, they provide a precise model of behavior of web software and is valuable for testing.

Once you generate this graph you can do whatever you want with it. You can do path testing, you can skip and test, you can generate test cases that correspond to identifying

various errors. So, testing based on atomic sections which in turn involves testing using ATGs and CIMs, it provides a nice mechanism to capture fundamental system level behavior of a server software. And please remember software of the presentation layer not the database layer. Such a testing, through examples, has proven to handle distributed integration of various websites, it can handle dynamically generated web pages to see if all the different ways of generating web pages can be done correctly.

It can handle operational transitions which are, which if you remember I told you were interrupts because somebody press the refresh button while a page was loading. Somebody press the back button while a page was loading, you can test for all these features of a server at the system level. The problem is that the graph CIM and ATG partly have to be generated manually. ATG graph, we do not know of an automatic way to generated by looking at the server side software. And this can be nontrivial, but once somebody is willing to putting that effort that in generate the graph then you give valuable information to test for all these properties of a web server, that is the point that is being made.

(Refer Slide Time: 28:42)

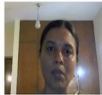


Some current open questions/problems

- There is no notion of data-flow based testing that is known for web applications. Static definition+use is difficult to determine.
- ATG cannot be automatically generated.
- Issues involving multiple users, session data, concurrency, dynamic integration etc. cannot be handled.

NPTEL

Page 46 of 49

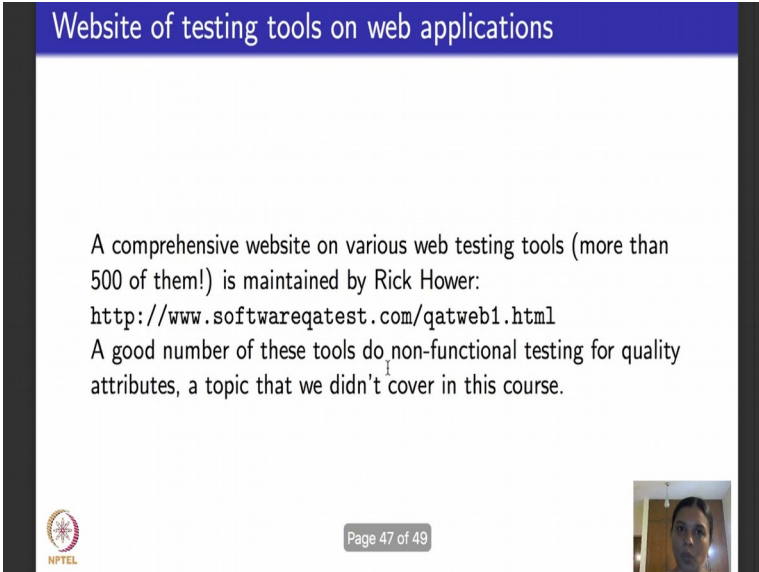


So, that is summarizes one technique for doing server side testing of web applications. So, just to recap what we did from the beginning, I introduced you to faults in websites. And then we looked at client side testing, one technique bypass testing. In this lecture I told you about graph based testing for system level testing of server code of web

software, what is left as far as web application testing is concerned. This area still very rich lot of open problems for example, if you consider what how does data flow graph model. If you see the ATG graph and CIM graph that we looked at models only control flow but at this system level, does not really talk about what happens to data flow. There is no clarity on data flow testing for system level web applications. People do not know how to do definition and use across dynamically generated web pages. So, that is very much an open problem that is left for working.

As I told you ATG cannot be generated automatically. Algorithms that will look at the server code on the presentation layer and generate ATG automatically would be very valuable. For example, if there are multiple uses if they session data this concurrency in the sense of multiple clients from different geographically located places handling, we still do not know how to generate graph models and test. All these are still rich in open problems that people could consider working on.

(Refer Slide Time: 30:10)



Website of testing tools on web applications

A comprehensive website on various web testing tools (more than 500 of them!) is maintained by Rick Hower:
<http://www.softwareqatest.com/qatweb1.html>
A good number of these tools do non-functional testing for quality attributes, a topic that we didn't cover in this course.

NPTEL

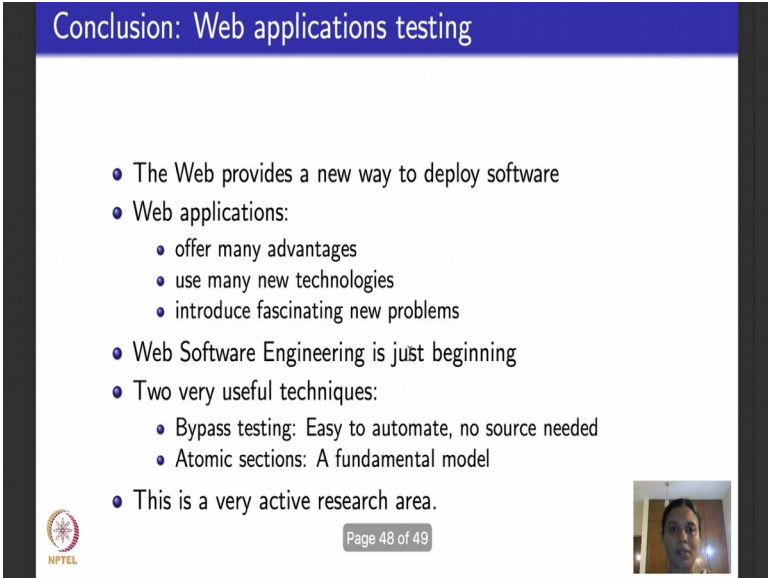
Page 47 of 49

Before I wind up this module on web applications testing, I would like to point to a nice website which maintains various web testing tools. In fact, if I know it write it as more than 500 different web testing tools. So, I just told you that there are several open problems and this website has almost 500 tools. So, you can imagine the richness of web application testing what we have done in these 3 lectures is probably to just scratch the surface of web application testing. This is the URL of this website, it is maintained this

by this gentlemen called Rick Hower. One thing to note before you access this website is that my focus on my lectures was on concentrating on system level functional testing of web application. I am testing the core functionality on the client side the first server side in terms of it does it do what it is supposed to do in terms of meeting it is functionality. That kind of techniques, testing techniques that I discussed with you does not involve things like performance testing, stress testing, low testing, which are non functional testing.

A lot of tools listed in this website talk about non functional testing of web applications. So, in case you are accessing this website, please remember that that something that we did not discuss in these 3 lectures on web applications.

(Refer Slide Time: 31:26)

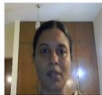


Conclusion: Web applications testing

- The Web provides a new way to deploy software
- Web applications:
 - offer many advantages
 - use many new technologies
 - introduce fascinating new problems
- Web Software Engineering is just beginning
- Two very useful techniques:
 - Bypass testing: Easy to automate, no source needed
 - Atomic sections: A fundamental model
- This is a very active research area.

NPTEL

Page 48 of 49





So, to conclude this module on web application testing. The web applications of a many advantages, they use several different technologies and as I told you there are still several fascinating open problems to work on. And I believe that is only the beginning of, if you want to coin a term called software engineering's specific to web software engineering. Research in it is still at it is beginning. What we looked at where 2 techniques bypass testing and graph models based on a atomic section. Bypass testing on the client side, atomic section on the server side, this is still a very active and rich area.

(Refer Slide Time: 32:04)

Some references

Here are some references for web applications testing.

- Sebastian Elbaum, Gregg Rothermel, Srikanth Karre and Marc Fisher II, Leveraging user-session data to support web application testing, in *IEEE Transactions on Software Engineering*, 31(3), 2005.
- Jeff Offutt, Ye Wu, Xiaochen Du and Hong Huang, Bypass testing of web applications, in *Proc. IEEE ISSRE*, 2004.
- Filippo Ricca and Paolo Tonella, Analysis and testing of web applications, in *Proc. ACM ICSE*, 25-34, 2004.
- Jeff Offutt and Ye Wu, Modeling presentation layer of web applications for testing, *Software and Systems Modelling*, 9(2), 257-280, 2010.



Page 49 of 49

So, here are some publications that I pulled out the material. From the first part which I did not do which is testing on the client side and on server side based on user session data. That you can find it in this paper by Elbaum, Rothermel, Karre and fisher it is an IEEE transactions on software engineering paper, about 12 years old. The thing that I taught you web pass testing bypass testing of web applications on the client side appeared in this paper by Offutt, Wu, Du and Huang in ISSRE 2004. This paper is another early paper that appeared in ICSE 2004 which talks about graph models of web testing the kind of things that I told you about 5 different transitions, between CIMs all those things you can find in this paper by Ricca and Tonella.

The server side testing that we discussed which is modeling the presentation layer as CIM and ATG. That was in the paper by Jeff Offutt and Wu which appeared in this general software and systems modeling in 2010. Feel free to pick up any of these papers if you want to know additional details or you could ask me in the forum if you have any specific questions if I note I will be able to answer it would be happy to do so. So in the next module will begin object oriented testing.

Thank you.

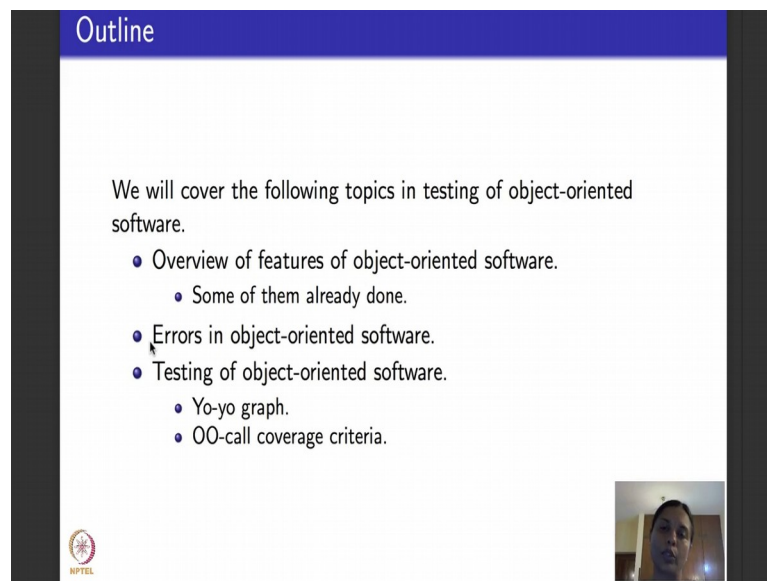
Software testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 48
Testing of Object-Oriented Applications

Hello again, we continue with week 10's lecture. Last three lectures we discussed about if I have a web application and I want to test for functionality at the system level where the client software and the server software is put together, what are the properties and techniques we can test for web applications.

Now moving on, what I would like to begin with this lecture is testing of object oriented applications. If you see object oriented programming languages, the most classical of them being Java or even C++ are extensively used and even, in fact, Android code could also be object oriented.

(Refer Slide Time: 00:58)



The slide is titled "Outline" in a blue header. The main content area is white and contains the following text and bullet points:

We will cover the following topics in testing of object-oriented software.

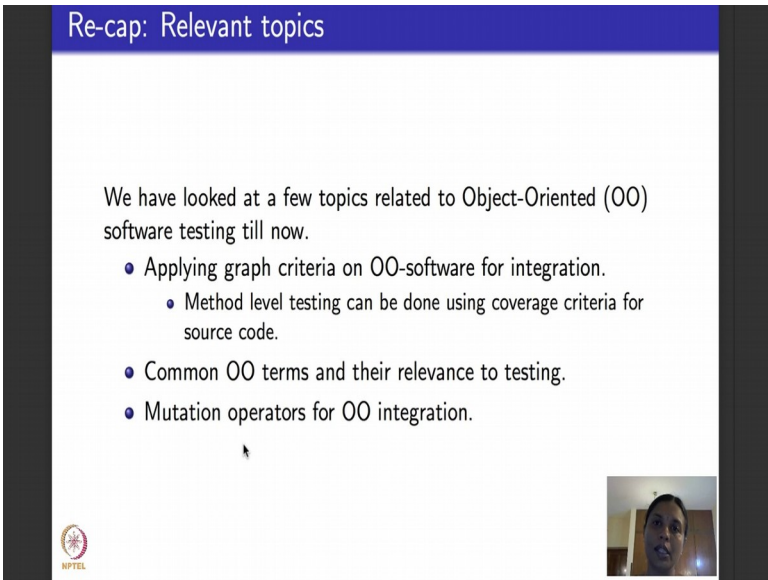
- Overview of features of object-oriented software.
 - Some of them already done.
- Errors in object-oriented software.
- Testing of object-oriented software.
 - Yo-yo graph.
 - OO-call coverage criteria.

In the bottom right corner of the slide, there is a small video feed showing a person's face. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.

So, we will see generic testing of object oriented applications over the next few lectures beginning with today. So, what is the outline that we are going to cover in the next few lectures? We will cover the following topics I will begin with giving overviews of features of object oriented software.

If you remember when we did mutation testing, I had already given you several features of object oriented software. If you missed that I urge you to go back and look at those lectures because I will not be recapping them in these lectures following which we will discuss about errors or anomalies that arise specific to features of object oriented software especially inheritance and polymorphism. And then we will discuss testing of object oriented software very unique graph models which are labeled Yo-Yo graph after the toy Yo-Yo, arise when testing object oriented software. So, we will see that model today and we will also define several call coverage criteria specific to object oriented software.

(Refer Slide Time: 01:52)



Re-cap: Relevant topics

We have looked at a few topics related to Object-Oriented (OO) software testing till now.

- Applying graph criteria on OO-software for integration.
 - Method level testing can be done using coverage criteria for source code.
- Common OO terms and their relevance to testing.
- Mutation operators for OO integration.

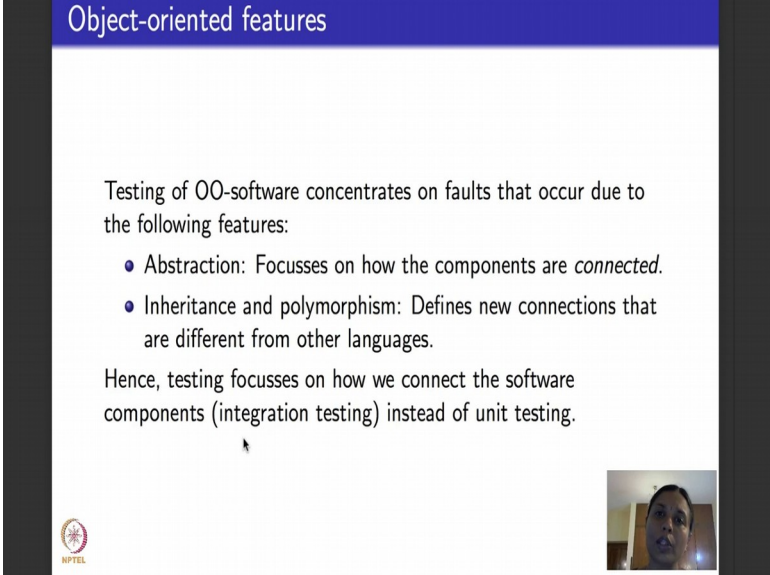
NPTEL

So, what have we done till now in all the lectures that we have covered when it comes to object oriented software. So, if you remember when we did integration testing, we specifically saw how to apply graph based coverage for object oriented software integration.

And in that, if you consider unit testing which means testing inside each method that can be done by using any traditional testing that we saw--- like graph based testing, logic based testing, all of them apply equally well to unit test. And then we also I told you we saw common object oriented terms and introduced them as they relevant to testing. To reiterate it was not meant to be a thorough introduction to object oriented programming, but only just a list of features as we would needed for testing and when we did mutation

operators, specifically integration testing mutation operators we saw operators for object oriented integration also.

(Refer Slide Time: 02:49)



The slide is titled "Object-oriented features" in a blue header. The main content area is white and contains the following text:

Testing of OO-software concentrates on faults that occur due to the following features:

- Abstraction: Focusses on how the components are *connected*.
- Inheritance and polymorphism: Defines new connections that are different from other languages.

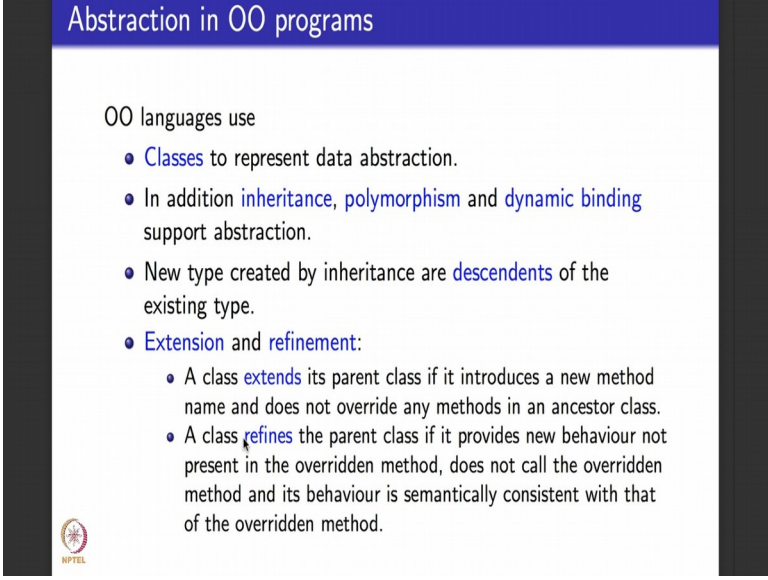
Hence, testing focusses on how we connect the software components (integration testing) instead of unit testing.

In the bottom left corner, there is a small circular logo with the text "NPTEL" below it. In the bottom right corner, there is a small video inset showing a person's face.

So, now what I will do is I will introduce and recap some of the relevant object oriented features that we will need for this lecture in the next couple of lectures. So, what do object oriented software do? Once a central features that object oriented programming provides is that of abstraction. So, abstraction let us you abstract the kind of information that you want typically as classes and you focus on how the components are connected, that various abstract components are connected. We also discussed features like inheritance and polymorphism when I introduced them last time. They basically define new connections that these abstractions offer and these are typically not found in of non object oriented languages.

So, our module as we will see for object oriented programming please remember we will not focus on testing inside a method what we are instead going to focus on is integration testing. I will show you the four levels of object oriented testing and also show you which part that we are going to focus on.

(Refer Slide Time: 03:54)



The slide has a blue header with the text 'Abstraction in OO programs'. The main content is on a white background with a black border. It lists concepts used in OO languages for abstraction. A small NPTEL logo is in the bottom left corner.

Abstraction in OO programs

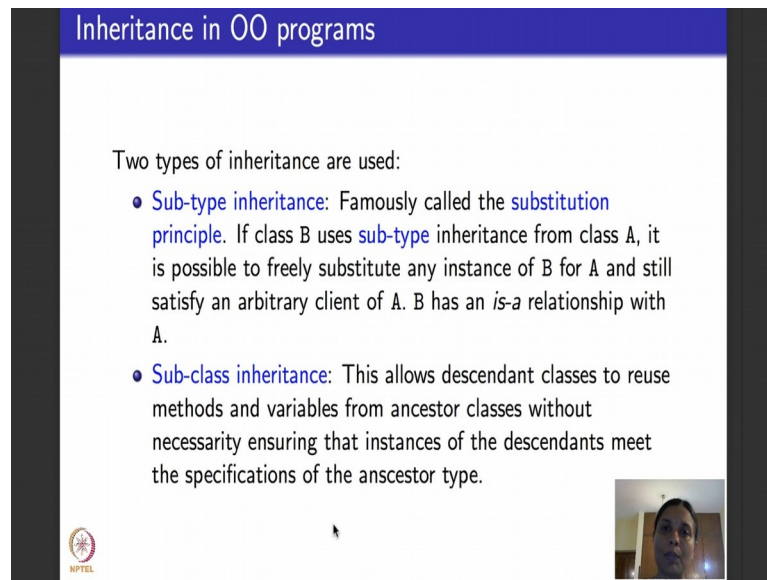
OO languages use

- **Classes** to represent data abstraction.
- In addition **inheritance**, **polymorphism** and **dynamic binding** support abstraction.
- New type created by inheritance are **descendents** of the existing type.
- **Extension** and **refinement**:
 - A class **extends** its parent class if it introduces a new method name and does not override any methods in an ancestor class.
 - A class **refines** the parent class if it provides new behaviour not present in the overridden method, does not call the overridden method and its behaviour is semantically consistent with that of the overridden method.

Before we move on what is abstraction and object oriented programs look like? The central entity in abstraction are that of classes, classes represent data abstraction. In addition to classes we have features like inheritance, polymorphism, dynamic binding, that also support abstractions, and how is a new type created? New type could be created by inheritance and as the word says, if it is a new type created by inheritance then whatever is created is called a descendant of the existing type of abstraction. Now there could be two kinds of new types that could be created. You could have extension or you could have refinement.

So, what is an extension? We say a particular class extends its parent class if it introduces a new method name and it does not override any method from its parent or ancestor. A particular class, on the other hand, refines a parent class if it provides new behavior that is not present in the overridden method and it also does not call the overridden method and in now along with that its behavior is semantically consistent with that of overridden method.

(Refer Slide Time: 05:07)



The slide is titled "Inheritance in OO programs" in a blue header. Below the title, it states "Two types of inheritance are used:". There are two bullet points: 1. "Sub-type inheritance: Famously called the substitution principle. If class B uses sub-type inheritance from class A, it is possible to freely substitute any instance of B for A and still satisfy an arbitrary client of A. B has an is-a relationship with A." 2. "Sub-class inheritance: This allows descendant classes to reuse methods and variables from ancestor classes without necessarily ensuring that instances of the descendants meet the specifications of the ancestor type." In the bottom right corner of the slide, there is a small video feed showing a person's face. In the bottom left corner, there is a small logo for NPTEL.

Inheritance in OO programs

Two types of inheritance are used:

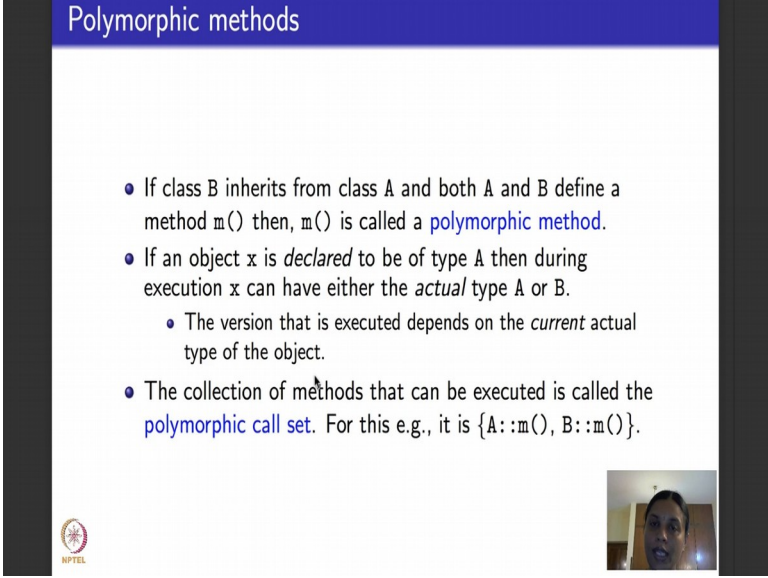
- **Sub-type inheritance:** Famously called the **substitution principle**. If class B uses **sub-type** inheritance from class A, it is possible to freely substitute any instance of B for A and still satisfy an arbitrary client of A. B has an *is-a* relationship with A.
- **Sub-class inheritance:** This allows descendant classes to reuse methods and variables from ancestor classes without necessarily ensuring that instances of the descendants meet the specifications of the ancestor type.

Moving on we considered two types of inheritance. Please remember all the terminologies that I am defining currently now is irrespective of the specific object oriented programming language that I use. These are generic object oriented features that. We will consider towards testing of object oriented software. So, there are two kinds of inheritance one is called the subtype inheritance, this is famously called substitution principle invented by this computer scientist called Barbara Liskov, who subsequently won the Turing award for this.

So, we say a particular class, let us say class B, uses subtype inheritance from class A, if it is possible to freely substitute any instance of B for a and still satisfy an arbitrary client for class A. So, we say B because it can be freely substituted for any A. So, we say B has what is called “is a relationship”. What do we mean by that? That any instance of class B is also an instance of class A. So, B can be freely substituted for class A. The next kind of inheritance is what is called subclass inheritance, where descendant classes reuse methods and variables from ancestor classes without necessarily ensuring that the instance of the descendant classes meet the specifications of the ancestor class.

So, two kinds of inheritance: subtype and subclass. Subtype means B is also class A, can be freely used like that, subclass means descendant classes reuse methods and variables, but they do not have to meet all the specifications of a parent class.


(Refer Slide Time: 06:47)



Polymorphic methods

- If class B inherits from class A and both A and B define a method $m()$ then, $m()$ is called a **polymorphic method**.
- If an object x is *declared* to be of type A then during execution x can have either the *actual* type A or B.
 - The version that is executed depends on the *current* actual type of the object.
- The collection of methods that can be executed is called the **polymorphic call set**. For this e.g., it is $\{A::m(), B::m()\}$.

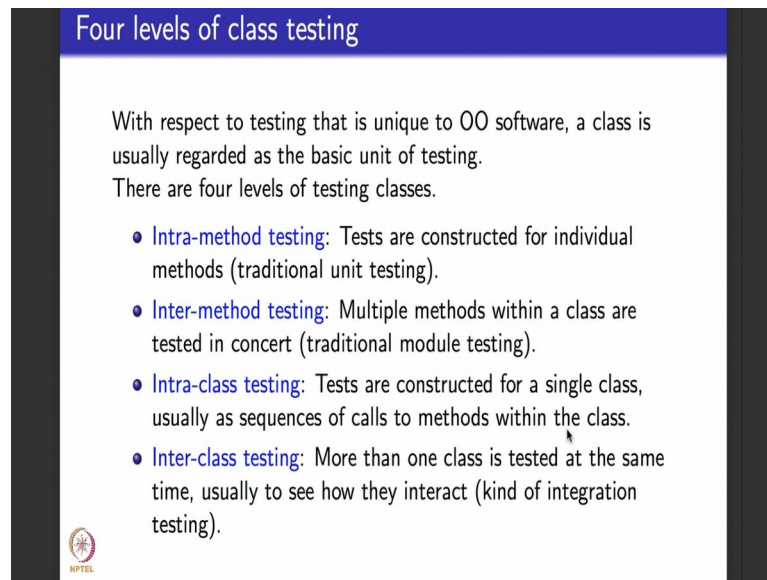
NPTEL



Now, just to recap what polymorphic methods are, because we would need this extensively today. We have already done this in one of the lectures, but I will recap it once again. So, consider two classes A and B and let us say class B inherits from class A and let us say both the classes A and B define a particular method m . So, m is defined in A and m is also defined in B and in addition B inherits from A. Such a method m in object oriented programming is called polymorphic method.

So, what happens now with polymorphic methods? Suppose there is an object X that is declared to be of type A then during execution, because B inherits from A the actual type can be A or B. Whichever the type that takes is completely dependent on the execution that happens right. This collection of methods which are polymorphic that can be executed when a particular object of a particular type is called is called a polymorphic call set. For example, in this case where there are two classes A and B and both define a method m , the polymorphic call set is $A::m()$, as it occurs in class A and m as it occurs in class B.


(Refer Slide Time: 08:03)



Four levels of class testing

With respect to testing that is unique to OO software, a class is usually regarded as the basic unit of testing. There are four levels of testing classes.

- **Intra-method testing:** Tests are constructed for individual methods (traditional unit testing).
- **Inter-method testing:** Multiple methods within a class are tested in concert (traditional module testing).
- **Intra-class testing:** Tests are constructed for a single class, usually as sequences of calls to methods within the class.
- **Inter-class testing:** More than one class is tested at the same time, usually to see how they interact (kind of integration testing).



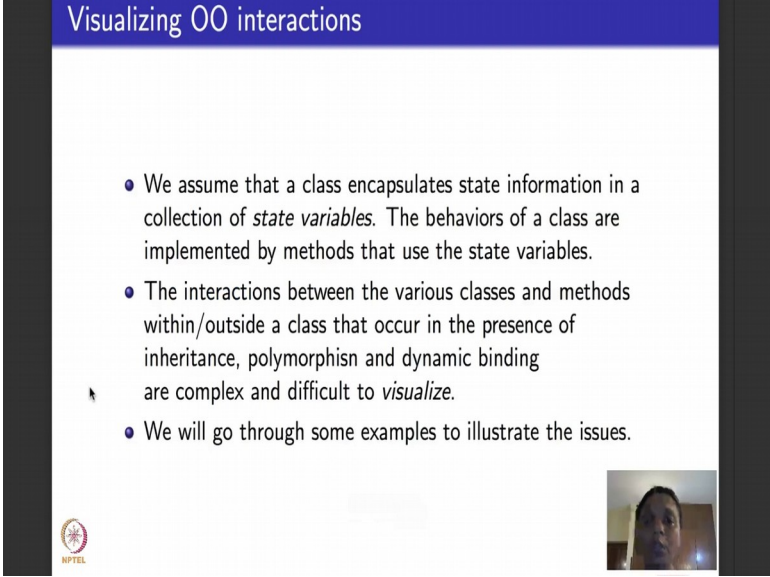
So, when while testing object oriented software we consider four levels of testing. We consider class as the basic unit of testing not a method as I told you when we consider a method as a basic unit of testing you could use any of the earlier coverage criteria that we learnt graphs, logic, mutation testing to be able to do method level testing. For us now focuses on a class level testing. So, with reference to class level testing for object oriented software, they could be four categories of testing. The first most elementary category involves intra method testing; that means, testing a particular method.

As I just told you use any applicable condition that we have learnt till now to test whether the method behaves as per its functionality, this is traditional unit testing. Moving up one level up from a method the next level of testing is inter method testing where interactions between methods are tested. This is traditional module level integration testing we have checked this when we did design integration testing based on graphs. In fact, we saw specific object oriented integration operators even for mutation.

The next is intra class testing that is testing within a class, what happens here? Tests are conducted for a single class that fixed to be tested at a particular point in time usually the tests are a sequence of calls to methods within a class. So, this class will define several methods a test case for the whole class will involve all the calling each of the methods, a select set of methods and so on. Finally, the full blown object oriented testing feature is a inter class testing where more than one class is put together and tested at the same time

usually to see how they interact. We will be considering inter class and intra class testing in these lectures.

(Refer Slide Time: 09:58)



The slide is titled "Visualizing OO interactions" in a blue header. It contains three bullet points:

- We assume that a class encapsulates state information in a collection of *state variables*. The behaviors of a class are implemented by methods that use the state variables.
- The interactions between the various classes and methods within/outside a class that occur in the presence of inheritance, polymorphism and dynamic binding are complex and difficult to *visualize*.
- We will go through some examples to illustrate the issues.

In the bottom left corner, there is a small circular logo with the text "NPTEL" below it. In the bottom right corner, there is a small video inset showing a person's face.

So, now before we move on, actual testing we will see a couple of lectures later. What I am going to tell you for the rest of this lecture and in the next lecture is what are the problems what are the difficulties, what are the anomalies, what are the faults that occur due to the specific object oriented features of inheritance, presence of polymorphic methods and dynamic binding. What happens in these cases? What are the issues that can come when we do inter class and intra class testing.

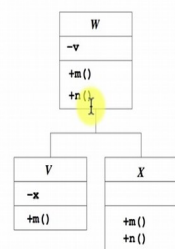
So, now before we move on, the most difficult thing is when you have a large many classes each defining its set of methods it is very difficult to visualize the actual sequence calls, the actual interactions that happen amongst these classes. I will show you examples of how difficult it is. So, we want to be able to do that. So, to do that we assume that a class encapsulates all the state information that I need. This is a traditional way, there is nothing related to object oriented software in any piece of program, what is the state of a program? State of a program defines the values of all its variables along with the location counter where the program decides.

So, in object oriented programming we assume that the class as an entity, encapsulates all the state information as a collection of state variables as it is done in other programming language. So, now, the behaviors of a class; how do they come? Class is a

static entity not executable. So, the behavior of a class is implemented by a set of methods that use these state variables. The interactions between various classes and methods, methods could be inside the same class or it could be outside a class will occur in the presence of all these features, inheritance, polymorphism, dynamic binding and those are the features we want to understand and visualize how the interactions happen.

(Refer Slide Time: 11:49)

Class hierarchy and method overriding: Example



```

1. void f(boolean b)
2. {
3.   W o;
4.   ...
5.   if(b)
6.     o = new V();
7.   else
8.     o = new W();
9.   ...
10.  o.m();
11. }

```

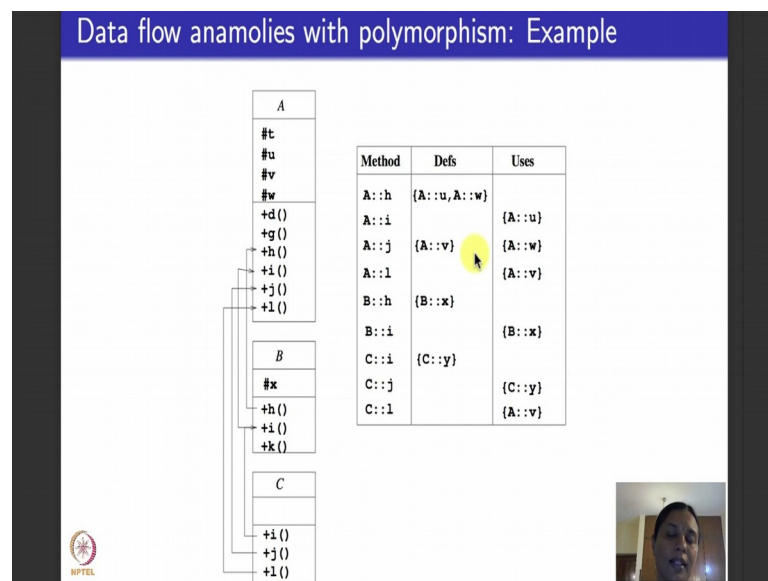
- V and X extend W, V overrides method m() and X overrides methods m() and n().
- -: attributes are private, +: attributes are non-private.
- The declared type of o is W, but at line 10, the actual type can be either V or W.
- Since V overrides m(), which version of m() is executed depends on the input flag to the method.

So, here is the first example consider this situation look at the left hand side of this figure. There are three classes W V and X and as per this picture V extends W, X also extends W. W has a private variable called small v and it has two methods m and n, V also has a method m which means this method m in V overrides the method m in W. Similarly the method m; methods m and n in X override the methods m and n in W. Minus sign indicates as I told you earlier the attributes are privates and plus sign indicates of the attributes are non private.

Now let us look at this piece of code on the right hand side. It is talking about some function f that takes a Boolean argument V and it begins by saying that the declared type of this object o is W and then there some piece of code dot dot dot read it as some piece of code and then let us say there is a new statement which says if this boolean variable b is true then you make o as what type, V. If this Boolean variable is false then you retain o to be of type W and later after some point let us say you call the method m for the object o. Now we know that the method m and V overrides the method m in W.

So, at this point in line number 10, we do not know which version of the method is called. Is Vs version of m called or its Ws version of m called that is not clear. It will purely be dependent on what the value of B is, this part here described by dot dot dot did it change B what happened to B. So, we need to know. So, usually this kind of analysis provides challenges because m overrides, the m in V overrides the m and W at this point in time we do not know which code of m got executed. So, this is the first kind of difficulty that we find in visualizing.

(Refer Slide Time: 13:51)



Moving on here is another kind of difficulty that you find with polymorphic methods this is a slightly bigger piece of program. What does it have? It has three classes A B and C, I have put them one below the other. A has how many variables four variables t, u, v and w, and it also has six methods d, g, h, I, j and l. Just for simplicity sake we have retained them a single letters they could be any method names. And then class B has one variable x. It has two three methods h, i and k. And what do these arrows indicate? These arrows indicate that the method h in B overrides the method h in A. Similarly the method i in B overrides the method i in A.


Now there is one more class C which has three methods I, j and l. And again there are arrows from this i pointing to the i of B, going by the same interpretation read this presence of this arrow as the method i in C overrides the method i in B, the method j in C overrides the method j in A, the arrow goes all the way up to class A and the method l in

C if you go trace the arrow and go up all the way, this method, overrides the method l in A. Is this clear that is what I have written here in the next slide.

(Refer Slide Time: 15:19)

Data flow anomalies with polymorphism: Example, contd.

- The root class *A* has four state variables and six methods, two descendents *B* and *C*. Methods of *A* are called in sequence.
- The state variables of *A* are protected, i.e., available to *B* and *C*.
- *B* declares one state variable and three methods, *C* declares three methods.
- *B::h()* overrides *A::h()*, *B::i()* overrides *A::i()*, *C::i()* overrides *B::i()*, *C::j()* overrides *A::j()* and *C::l()* overrides *A::l()*.
- Data flow anomaly: Suppose an instance of *B* is bound to an object *o* and a call to *d()* is made. *B*'s version of *h* and *i* are called, *A::u* and *A::w* are not given values, and thus the call to *A::j* can result in a data flow anomaly.



The root class *A* has four state variables six methods we saw that right, four state variables six methods. It also has two descendant classes *B* and *C* as I told you. Methods of *A* are called in sequence which means what you assume that in the code of there is some piece of code in a where method *d* calls method *g*, which in turn calls method *h*, which in turn calls method *i*, which in turn calls method *j* and *j* in turn calls method *a* that is what I have written here.

Now the state variables away as you can see in the notation are protected which means what, they are available to the descendant classes *B* and *C*. *B* declares one state variable three methods, *C* declares three methods. And we discussed this overriding which method of *B* overrides which methods *A*, which methods of *C* overrides which method of *A* and *B*.

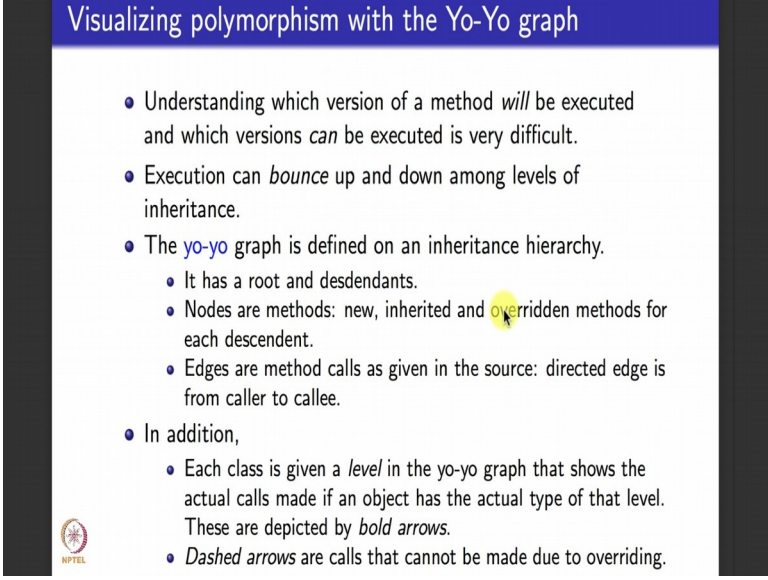
Now, consider a situation like this. What does this say? This says how to read this table this says in the method *h* as present in class *A*, two variables are defined *u* and *w*. And the variable *u* that is defined in the method *h* for *A* is used by the method *i* for *A*. Similarly the variable *v* is again defined in the method *j* present in *A* and method *j* also uses the variable *w*. The method a variable *v* defined in *j* gets used by the method *i* in *A*. Now *B* also defines a variable *x* in its method *h* *B* uses the variable *x* in its method *i*; *C*

similarly defines variable y in its method i and uses variables y and v in its method g and f. Is this clear?

Now I claim that because there are these polymorphic methods i and k and h and the j and l there is a problem with dataflow. How do you understand the problem with data flow? Here is the anomaly, suppose an instance of B is bound to an object o and let us say a call to d is made, a call to d is made right. Now B's d, what will d do - d will call g g will call h, but which version of h is called, B's version of h is called right. B's version of h is called which means what - B's version of h and B's version of i is called. If B's version of h and i is called then you look at it here. B's version of h, what is the variable that it defines it defines the variable x, B's version of i defines nothing. But then what do we want we finally, want A j because a in sequence I told you right g calls g sorry d calls g which in turn calls h and i, but h and i are the B's versions that are called and I further calls, what does j need ? j needs A w because it is going to use A w, but as version of I and h were not called only B's version of h and i were called and w was not defined B's version of h defines only x, B's version of I defines nothing.

So, that is what is said here B's version of h and I are have called A u and A w are not given values because only as version of h gives them values which means what the call to a j can result in a dataflow anomaly. It does not present at all. So, how will it use W there will be a problem. So, these are the kind of faults that we want to recover, identify and test for when we do object oriented programming.

(Refer Slide Time: 19:15)



Visualizing polymorphism with the Yo-Yo graph

- Understanding which version of a method *will* be executed and which versions *can* be executed is very difficult.
- Execution can *bounce* up and down among levels of inheritance.
- The **yo-yo** graph is defined on an inheritance hierarchy.
 - It has a root and descendants.
 - Nodes are methods: new, inherited and overridden methods for each descendent.
 - Edges are method calls as given in the source: directed edge is from caller to callee.
- In addition,
 - Each class is given a *level* in the yo-yo graph that shows the actual calls made if an object has the actual type of that level. These are depicted by **bold arrows**.
 - **Dashed arrows** are calls that cannot be made due to overriding.

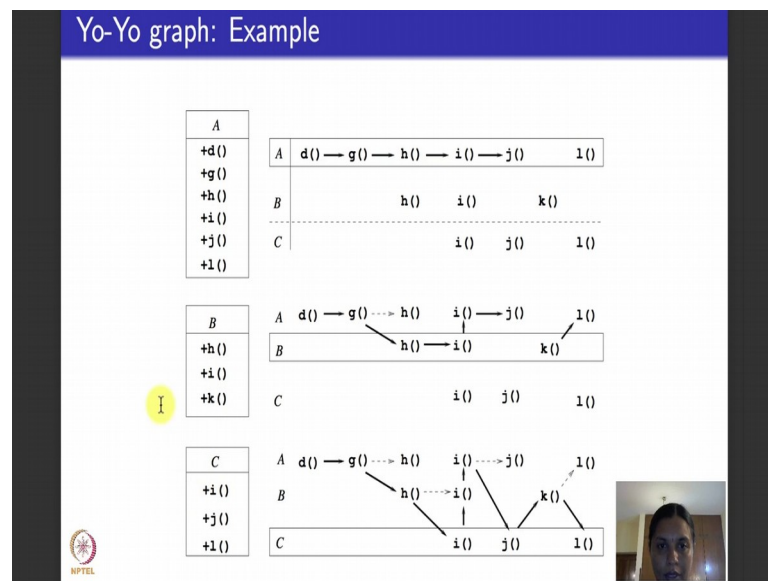
Now one last example before we move on where I introduced Yo-Yo graph formally to you. So, Yo-Yo graph deals with a problem of which is the version of the method that will be called. It is a very difficult problem to understand because the methods that can be called can bounce up the calls of the methods can bounce up and down like in a Yo-Yo that is what we will illustrate. So, what is a Yo-Yo graph? You all might be familiar with a Yo-Yo toy right this is string and then this top like thing where you can rotate the string and when you release the rotating entity from the string, it bounces up and down up and down right.

So, what we trying to say is that the method call graphs across classes can bounce up and down like a Yo-Yo. We will illustrate it for the example that we saw in this slide. So, before that what is a Yo-Yo graph. Yo-Yo graph is more like a tree it has a designated vertex called a root and then it has some descendants. What are the vertices of the graph? Vertices of the graph are not statements or anything like that, they are individual methods because we are focusing on method calls we focusing on call graph that deal with methods. So, nodes are methods, methods could be new methods, inherited ones or overridden methods for each of the descendant.

What are the edges? As I told you edges are the method calls that are given in the source code. A directed edge is present from the caller to the callee. In addition we have two other things there is a level that is given to each class in the Yo-Yo graph that shows the

actual calls made for an object if an object has a particular type of that level these are depicted by bold arrows. The Yo-Yo graph also has dashed arrows which are lighter which talk about the calls that cannot be made because the methods have been overridden. So, for the example that is given here, if you remember this method, class A six methods class B three methods class C three methods here is how the graph looks like.

(Refer Slide Time: 21:18)



So, I have just done the left hand side here that I am tracing with the cursor, I have just copied the same thing, but I have omitted the arrow is corresponding to overriding just not to clutter the figure too much. What have I depicted here? Again it is the same information: classes A, B and C are given here this is a list of methods that are present in the class A, this is list of methods that are present for class B and here is the list of methods that are present for class C. As I told you methods and class A call each other in sequence from top to bottom, which means d calls g, g calls h, h in turn calls i, i calls j. There is no problem everything works fine because these are all methods within the class A and the calls are absolutely fine.

But let us say you get this kind of visualization, what happens here? Method d is called in A, d calls g, g calls h, but which version of h B's version of h and B's version of h in turn calls B's version of i which calls A's version of i and then calls j and independently k to l call is present. Now if you see there is a d-link because I do not know what to do.

now this makes it even worse. So, say we start similarly we say d cause g, g calls h and h does not even call B's version it called C's version of I and C again calls B's version of i which in turn calls as version of i and As version of i now goes all the way down and calls j's C which calls B's k which calls l C and so on.

So, you can see this you can think of as a Yo-Yo unrolling itself, this is a Yo-Yo bouncing back, this is a Yo-Yo bouncing back and unrolling itself. So, the pattern of calls can grow up and down like a Yo-Yo. The dashed lines as I told you here are calls that cannot be made due to overriding, but if they had been made it could have been anomaly free that is what we are trying to say here.

(Refer Slide Time: 23:25)

Yo-Yo graph: Example

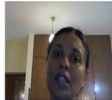

- A's implementation: d() calls g(), g() calls h(), h() calls i() and i() calls j().
- B's implementation: h() calls i(), i() calls its parent's (A's) version of i() and k() calls l().
- C's implementation: i() calls its parent's (B's) version of i() and j() calls k().

So, what I explained to you is what is written here. In A's implementation there is the sequence of method calls, in B's implementation there is the sequence of method calls h calls i and i calls its parent version of i as I told you here right I calls its parent version, parent is A so it calls its parent version of i and k calls l. In C's implementation I calls this I of C calls its parent version of i which is B's i which in turn calls A's i and then the call to j is made. So, the calls of methods go flat, bounce up, bounce down like a Yo-Yo hence the term Yo-Yo graph.

(Refer Slide Time: 24:06)

Yo-Yo graph: Example



- Top level: A call is made to method `d()` through an object of actual type `A`.
 - This sequence of calls is simple and straightforward.
- Second level: Object is of actual type `B`. When `g()` calls `h()`, the version of `h()` defined in `B` is executed. The control then continues to `i()` in `B`, `i()` in `A` and then to `j()` in `A`.
- Third level: Object is of actual type `C`. Control proceeds from `g()` in `A`, to `h()` in `B` to `i()` in `C`, then, to `i()` in `A` and `B` etc— exhibiting a yo-yo effect.



Now, if the top level call is made then there is absolutely no problem. For second and third level you could have data flow anomalies as we saw in this example. So, this is another kind of visualization that we will like to work on.

(Refer Slide Time: 24:17)

Next lecture: Categories of inheritance faults and/or anomalies.



I will stop here for this lecture. In the next lecture I will discuss with you about specific faults and anomalies that can occur due to inheritance and polymorphism.

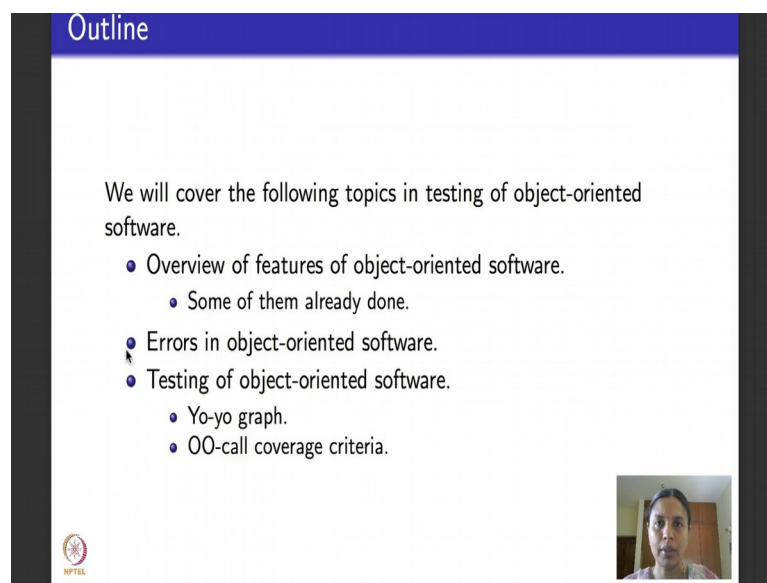
Thank you.

Software testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 49
Testing of Object-Oriented Applications

Hello again, this is a last lecture of week 10. Last time I had introduced to you to testing of object oriented applications.

(Refer Slide Time: 00:21)



The slide is titled "Outline" in a blue header. The main content area is white and contains the following text and list:

We will cover the following topics in testing of object-oriented software.

- Overview of features of object-oriented software.
 - Some of them already done.
- Errors in object-oriented software.
- Testing of object-oriented software.
 - Yo-yo graph.
 - OO-call coverage criteria.

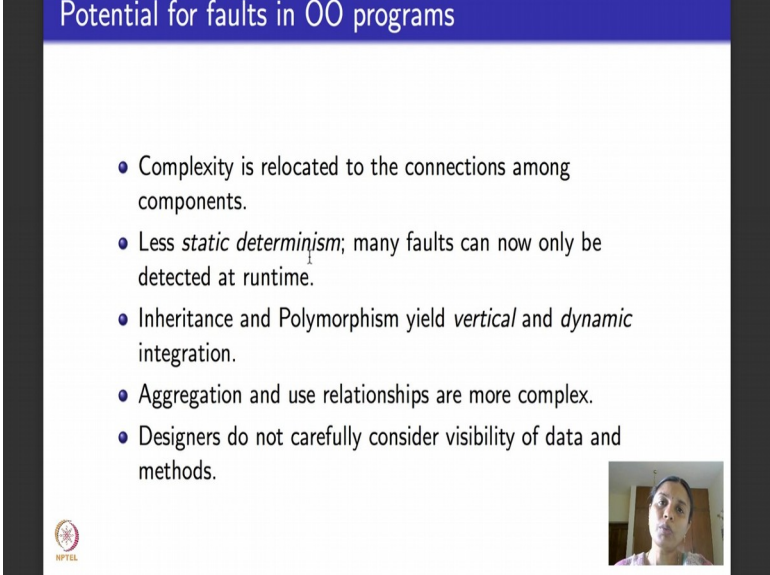
In the bottom left corner, there is a small circular logo with the text "IITEL" below it. In the bottom right corner, there is a small video inset showing a person's face.

This was an outline of the talks that I had discussed with you. We will look at an overview of some of the features of object oriented software we already done some in the context of mutation testing for object oriented integration. And then we said we will discuss about what are the kinds of faults or errors that will come an object oriented software. We will follow it up with testing of object oriented software. Yo-yo graph is something that we already looked at in the last module.

And then in the next lecture I will ended with object oriented call coverage criteria. Today what I am going to focus on is what are the errors in object oriented software. We already done overview in of object oriented features. So, I had introduced you to inheritance, polymorphic methods and what is the class hierarchy, what could be the various anomalies with object oriented testing and we had ended last class by looking at



this graph called Yo-yo. Now moving on what I am going to do today's lecture is focus on faults or errors that comes specifically related to object oriented programs.

(Refer Slide Time: 01:22)



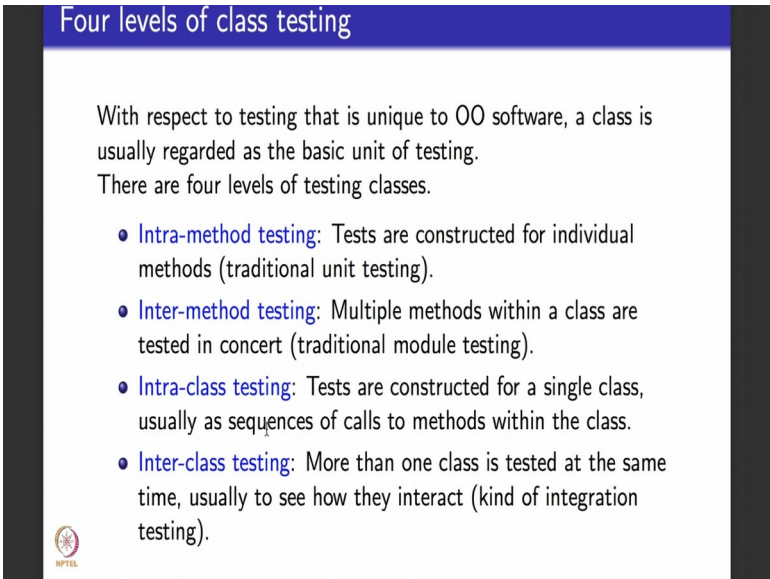
Potential for faults in OO programs

- Complexity is relocated to the connections among components.
- Less *static determinism*; many faults can now only be detected at runtime.
- Inheritance and Polymorphism yield *vertical* and *dynamic* integration.
- Aggregation and use relationships are more complex.
- Designers do not carefully consider visibility of data and methods.



Specifically with reference to the object oriented features that we have seen till now. Now what will be the complexity or the main source of errors? The main source of errors as I told you because we are focusing on which level of testing if you go back to this slide that I had looked at, we are focusing here we are focusing on intra class and inter class.


(Refer Slide Time: 01:41)



Four levels of class testing

With respect to testing that is unique to OO software, a class is usually regarded as the basic unit of testing.
There are four levels of testing classes.

- **Intra-method testing:** Tests are constructed for individual methods (traditional unit testing).
- **Inter-method testing:** Multiple methods within a class are tested in concert (traditional module testing).
- **Intra-class testing:** Tests are constructed for a single class, usually as sequences of calls to methods within the class.
- **Inter-class testing:** More than one class is tested at the same time, usually to see how they interact (kind of integration testing).

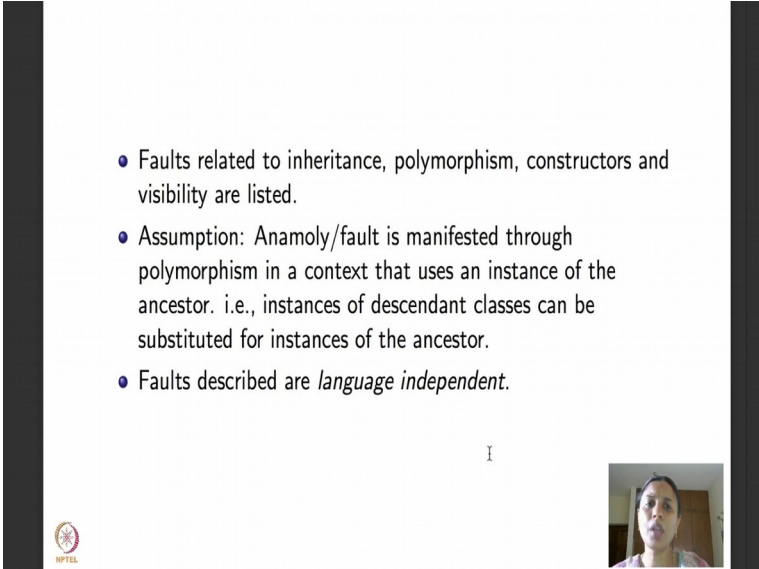


So, which means what we are focusing on various components and how they are going to interact. So, our complexity will or focus will be on the connections amongst these components. The problem with object oriented software as we have seen till now is that the amount of static determinism is very less.

In the sense that suppose I make a method call another method, or let us say I make a function call and other function and see I know that this is how the call is going to happen nothing is going to be different, whereas that is not true for an object oriented programming language. For method calls and other methods, the method that is being called can be overridden, the method that is called, link can be overridden, one of these methods could be polymorphic, all kinds of interactions can happen. Nothing is statically determined. Lot of things depend on dynamic execution.



As I told you the complicated features that we recapped reasonably well like inheritance, polymorphism, they yield vertical integration dynamically changing integration and there are aggregation and use relationships that also add to the complexity, and typically designers when they design and capsulation and abstraction they do not consider visibility on data and methods because they consider it as a tester's problem.

(Refer Slide Time: 03:05)



- Faults related to inheritance, polymorphism, constructors and visibility are listed.
- Assumption: Anamoly/fault is manifested through polymorphism in a context that uses an instance of the ancestor. i.e., instances of descendant classes can be substituted for instances of the ancestor.
- Faults described are *language independent*.

I

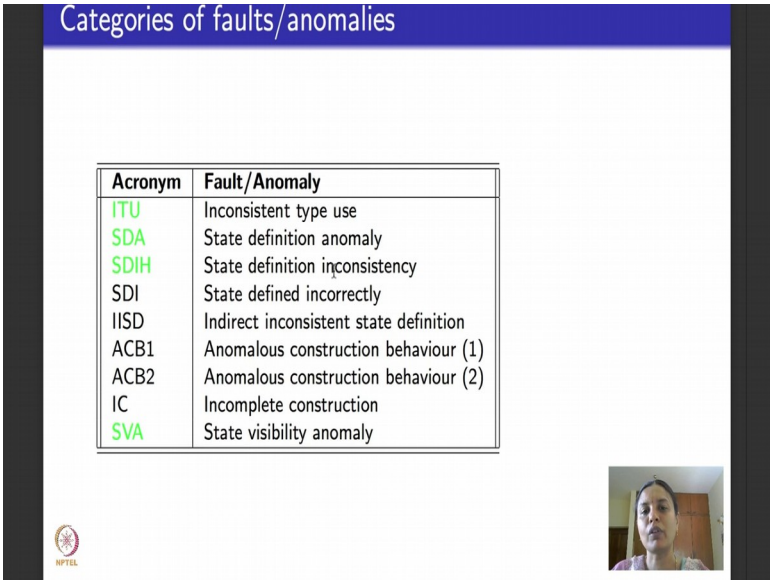


So, now what we are going to look at in these lectures just completely understand some of the faults that come because of these features: inheritance, polymorphism, constructors, visibility and so on.

While we understand these faults we will make one assumption will assume that the fault are anomaly is manifested through polymorphism in a context that uses in instance of ancestor. That is instances of descendants descendant classes can be substituted for instances of an ancestor. Another thing that I would like to tell you is that the faults that I have described here are completely language independent. It is not like these are exclusive only to java and they cannot be used for C++, they are generic faults that can arise in object oriented languages.

That use these features. So, they are independent of a particular programming language.

(Refer Slide Time: 04:02)



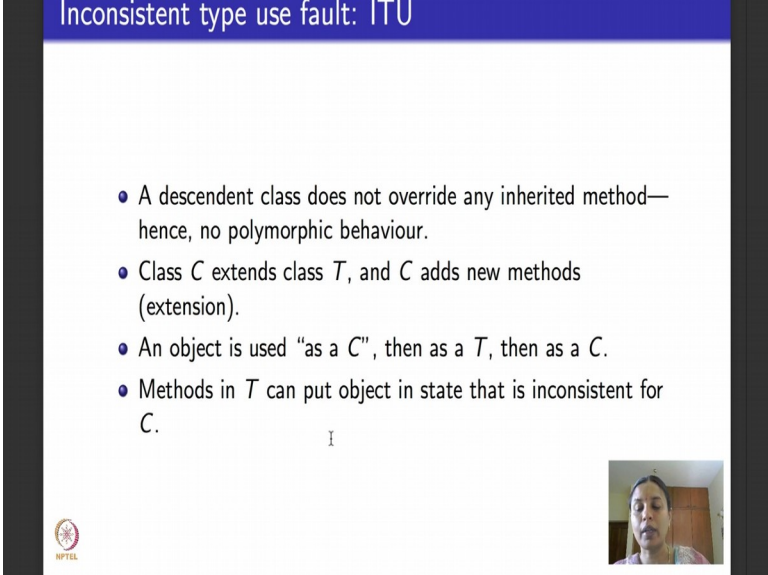
Acronym	Fault/Anomaly
ITU	Inconsistent type use
SDA	State definition anomaly
SDIH	State definition inconsistency
SDI	State defined incorrectly
IISD	Indirect inconsistent state definition
ACB1	Anomalous construction behaviour (1)
ACB2	Anomalous construction behaviour (2)
IC	Incomplete construction
SVA	State visibility anomaly

So, here is a table of the faults and anomalies that are commonly listed for method level, interclass and intra class level testing of object oriented applications. Do not worry too much about remembering these acronyms. They are just listed for convenience because its useful to use them in slides subsequently and while talking about it you do not have to memorize them. So, we will understand about the nine faults, 9 kinds of faults are anomalies are listed here I will explain about 5 of them in detail with examples illustrating what is the kind of fault or anomaly.

The remaining 2, 3 of them, I will just give an overview and I will point you to literature at the end of these slides where you could look up for additional information. So, we will look at inconsistent type use state definition can be anomalies or inconsistent or incorrect. We look at indirect and inconsistence state definition when it comes to

constructors, there could be 2 different kinds of anomalies behaviors. I have decided to skip these 2 ACB 1 and ACB 2 because we have not really looked at constructors, class constructors in detail. So, I will skip these 2, but I will tell you in detail about the other things. We will also do state visibility anomaly.

(Refer Slide Time: 05:22)



The slide has a blue header with the text "Inconsistent type use fault: ITU". Below the header, there is a white area containing a bulleted list of four points. In the bottom right corner of the slide, there is a small video inset showing a person's face. In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.

- A descendent class does not override any inherited method—hence, no polymorphic behaviour.
- Class *C* extends class *T*, and *C* adds new methods (extension).
- An object is used "as a *C*", then as a *T*, then as a *C*.
- Methods in *T* can put object in state that is inconsistent for *C*.

So, of the first one in the list is inconsistent type use fault, ITU. So, what is the cause of this fault? The cause of this fault is because a descendent class does not overwrite any inherited method? So, there is no polymorphic method polymorphic behavior because nothing is overridden everything is static. So, you could think; what is the problem? So, will understand it with an example. For example, consider a case where class *C* extends a particular class say *T* and *C* also adds new methods an object sometimes is used as a *C* sometimes is used as *T* and sometimes is used as a *C* again. Now methods in *T* can put this object in a state that is inconsistent for *C*. That is why it is called inconsistent type use fault. What do we mean by a state? We mean by a state in any programming language including object oriented programming language we mean the values of all the variables. So, this last point you must read it as methods can change the sub values of some of the variables which results in object going into a different kind of state.

(Refer Slide Time: 06:32)

Inconsistent type use fault: Example

- Class Vector is a sequential data structure that supports direct access to its elements.
- Class Stack uses methods inherited from Vector to implement the stack.

```
classDiagram
    class Vector {
        array
        +insertElementAt()
        +removeElementAt()
    }
    class Stack {
        +pop():Object
        +push():Object
    }
    Stack --> Vector : call
```

```
s.push("string1");
s.push("string2");
s.push("string3");
dumb(s);
pop();
pop();
pop(); //Stack is empty

void dumb(Vector v)
{
    v.removeElementAt(v.size()-1);
}
```

So, here is an example let us state we have these 2 classes there is a class called vector which uses a sequential data structure called array and it supports direct access to all its elements. It has 2 methods one method thus inserting an element at a particular point.

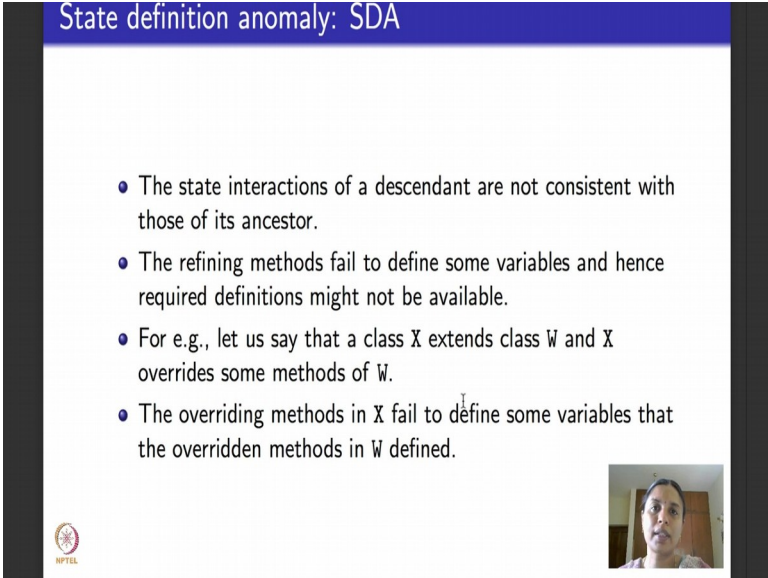
And it also does removing an element at a particular point. There is a class stack. What is class stack do? It does the normal operations of a stack it pushes or it pops an object. So, to pop an object it calls the method remove element from the inherited from the class vector to push an object it calls the method insert element inherited from the class vector. So, here is a abstract piece of code again. Like all the examples that we saw in the course this course is this code is not complete, but only a snippet of it is given may helping you to understand the code issue.

So, let us say there is there are 3 operations that happen in this code, 3 push operations. String 1, string 2 and string 3 are pushed using this method in the class stack and let us say, at some point this method dumb is called. Let us go and look at the code of dumb. What does dumb do? dumb removes an element at v of size minus one, which means in this case it removes the middle element. So, after a call to dumb is made, when it returns, out of the 3 entities that we have pushed into s, the stack s, only 2 of them are available. Now what look at these series of calls of pops, first pop after the 2 elements that are available in s.

The one that is available top most gets popped out, second pop there is only one left it gets popped up now for the third one there is nothing to pop. The stack is already empty. Why is there is nothing to pop even though there were 3 different pushes? That is because there was this call may to a method dumb which also use to remove element which the class stack did not know about. Because it use remove element one of the elements went away the class stack assume that because there were as many pushes is there pops you would have elements for to do 3 pops, but elements were not there because of this.



So, this resulted in an inconsistent assumption on the state of s here it assumes that there were 3 elements present whereas, actually only 2 elements were present.

(Refer Slide Time: 09:06)



State definition anomaly: SDA

- The state interactions of a descendant are not consistent with those of its ancestor.
- The refining methods fail to define some variables and hence required definitions might not be available.
- For e.g., let us say that a class X extends class W and X overrides some methods of W.
- The overriding methods in X fail to define some variables that the overridden methods in W defined.

So, this is a classical example of how inconsistent type use can occur the next kind of fault that I want to explain is what is called state definition anomaly SDA. What happens here? The state interactions of a descendent are not consistent with those of its ancestor. What do we again we mean by state interactions? State interactions means operations that involve changing the state which means operations that change one or more variables that are present in the state.

(Refer Slide Time: 09:56)

State definition anomaly: Example

```
graph TD
    W[W] --> X[X]
    X --> Y[Y]
```

W:
v
u
m()
n()

X:
x
n()

Y:
w
m()

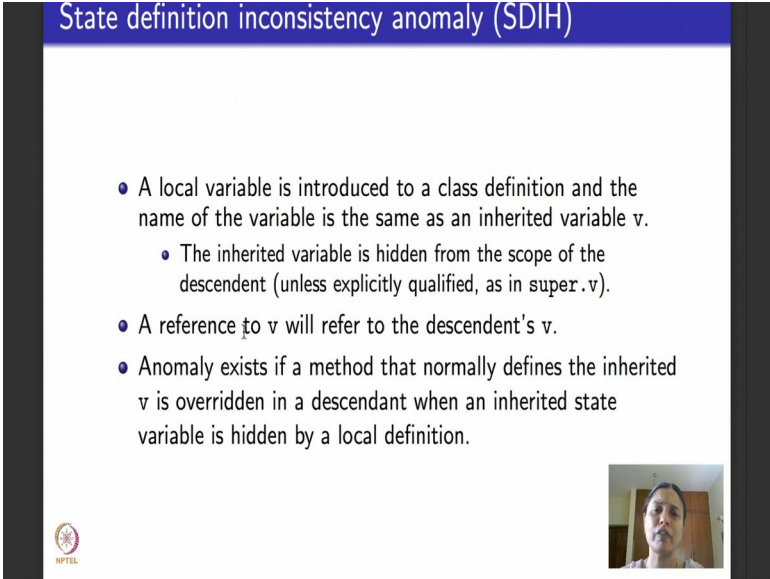
W::m() defines v and W::n() uses v
X::n() uses v
Y::m() does not define v

For an object of actual type Y, a data flow anomaly exists and results in a fault if m() is called, then n().

So, the refining methods fail to define some variables and the required definitions that you need may not be available. We will see an example that explains this in detail. Let us say there is a class X that extends W and along with the extension X also overrides some of the methods of W. The overriding methods in X fail to determine some variables that the overridden methods in W actually define. Here is an example let us say there are 3 classes W X and Y. So, W defines 2 variables u and v and has 2 methods m and n. X has one variable small x and has one method n and y has one variable W small W and has one method m.



So, W this method m and W defines v and this method n in W uses v let us say. Some piece of code, I have not given the actual piece of code and let us say the method X and n also uses v and let us say the method Y in m does not define v at all. So, now, for an object of actual type y there will be a data flow anomaly if a fault in m, if m is called and then n. Why because if m is called and then m Y m does not define v and W m is the one that actually defines v. Y m does not define v and X n tries to use v, but there will be no v available for X n to use. So, this results in a variable not being defined which means a state a part of the state not being defined and hence is called state definition anomaly.

(Refer Slide Time: 11:15)



State definition inconsistency anomaly (SDIH)

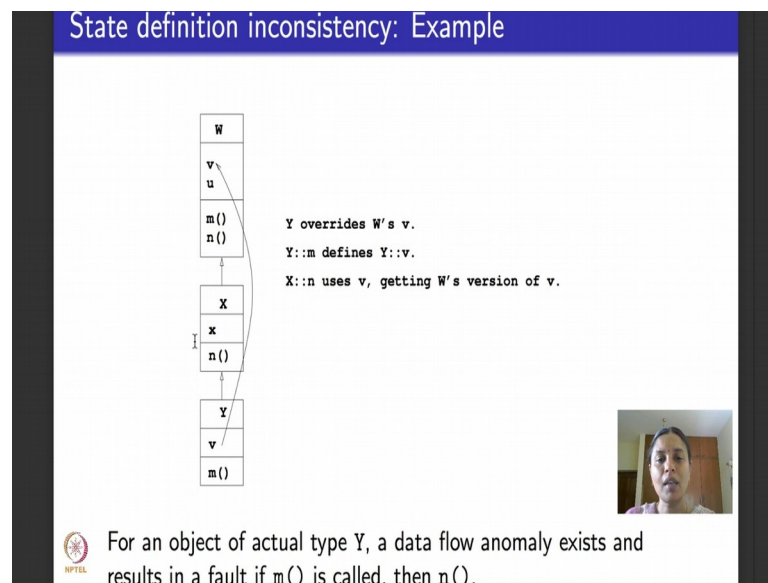
- A local variable is introduced to a class definition and the name of the variable is the same as an inherited variable *v*.
 - The inherited variable is hidden from the scope of the descendent (unless explicitly qualified, as in `super.v`).
- A reference to *v* will refer to the descendent's *v*.
- Anomaly exists if a method that normally defines the inherited *v* is overridden in a descendant when an inherited state variable is hidden by a local definition.

The next kind of fault is state definition inconsistency fault SDIH. Here what happens, a local variable is introduced in to a class definition and the name of the variable is the same as the name of the inherited variables. So, there is some amount of inconsistency that can happen because 2 different variables share the same name the inherited variable is hidden from this scope of descendant.

Of course you if you explicitly use this keywords `super` and refer to the variable *v* then it is not hidden, but otherwise you can assume that the inherited variable is hidden from the scope of the descendant. Now a reference to *v* will actually referred to the descendant's *v* because that is where it is rewritten. An anomaly can exist if a method that normally defines the inherited variable is overridden in a descendant, when an inherited state variable is hidden by a local definition. So we will see again in an example.

(Refer Slide Time: 12:12)





So, consider the same kind of situation we have 3 classes W X and Y. W has 2 variables u and v, 2 methods m and n. X has a variable small X and a method n, Y has a variable small v and a method m. Now the catch is, the Y, the class Y overrides, the class Y overrides this W's v. So, which means what these 2 v is the same, but this v overrides this v. Now the method m of y defines this variable y now X in this method n uses v, but whose version will it get? It will get W's version, is that clear. So, now, for an actual type of object actual type y there can be a data flow analysis if m is called an then n is called because X is actually expecting W's version whereas y is already overridden it. So, it the v that is sent is basically not the expected v.

(Refer Slide Time: 13:20)

State visibility anomaly (SVA)

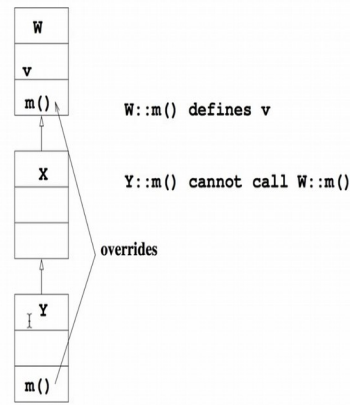
- Consider a class W , which is an ancestor X and Y : X extends W and Y extends X .
- W declares a private variable v , v is defined by $W::m$.
- Y overrides $m()$, and calls $W::m()$ to define v .
- This causes a data flow anomaly as v is private for W .



So, this state part corresponding to that v is inconsistent. So, the next fault state visibility anomaly. Consider a class W which is an ancestor of X and Y . X extends W , Y also extends X . Now say W declares a private variable v in a method m , Y overrides that method m calls $W::m$ to define v . Now this causes data flow anomaly because the call for y is not well defined. Why, because v is private for W right? So, which means the visibility or the access level of v is not consistent the way Y and W want to use it, that is what is wrong.

(Refer Slide Time: 13:57)

State visibility anomaly: Example



W

v

$m()$

$W::m()$ defines v



X

$Y::m()$ cannot call $W::m()$

overrides

Y

$m()$

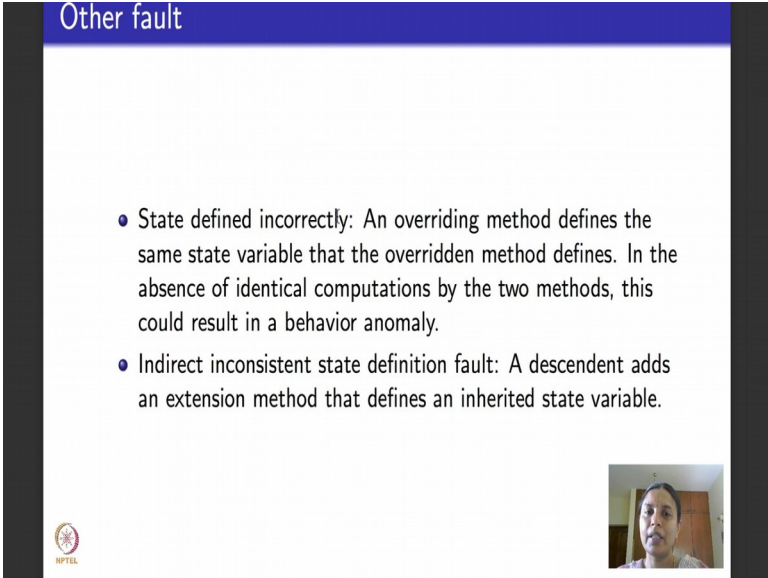


So, this is what is explained in the previous slide 3 classes W X and Y. Y has a private variable v and defined by the method W has a private variable v sorry, defined by the method m and y has a method m that overrides this method. So, y m cannot call W m because of W is I mean the variable v is private to w.

So, that results in the visibility of v not being there. So, that is what is called state visibility anomaly. There are several other faults that were listed in the table that I presented to you earlier. For a second I will go back to that table. So, these where the faults. Why have I colored them these 4 green, because I gave you examples for these 4 that is why these 4 were colored green. We will also tell you what is state defined in consistently an indirect inconsistent state definition is as I told you we will not discuss the 3 constructive faults.

Because I dint introduced a notion of class constructor in detail to you. So, I will skip it in my lectures, but I will give you references where you can learn more from it. So, the 2 pending things without examples are state defined incorrectly and indirect inconsistent state definition.

(Refer Slide Time: 15:16)



The slide is titled "Other fault" in a blue header. It contains two bullet points:

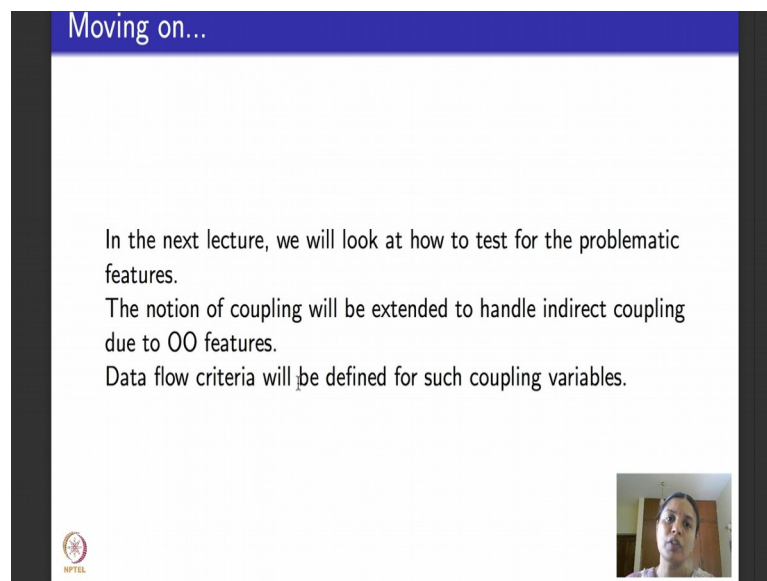
- State defined incorrectly: An overriding method defines the same state variable that the overridden method defines. In the absence of identical computations by the two methods, this could result in a behavior anomaly.
- Indirect inconsistent state definition fault: A descendent adds an extension method that defines an inherited state variable.

In the bottom right corner, there is a small video inset showing a person speaking. In the bottom left corner, there is a small logo for NPTEL.

So, we will look at that. State defined incorrectly, what happens here? An overriding method defines the same state variable that the overridden method also defines. So obviously, 2 variables defined by 2 different methods, one could say values is one, the other could say values the other state will be defined incorrectly.

In the absence of identical computations, if the 2 methods do identical computations fine, but let us say 2 methods do different things they could highly likely that they will come up with different values for the same variable *v*. So, the state gets defined incorrectly. The next is indirect inconsistent state definition fault this is when happens when a descendant adds an extension method that defines an inherited state variable. The reference that I will be pointing you to you will find examples for these also. Feel free to look at them in you can ping in the forum if you do not understand anything or you have doubts. .

(Refer Slide Time: 16:06)



Moving on...

In the next lecture, we will look at how to test for the problematic features.
The notion of coupling will be extended to handle indirect coupling due to OO features.
Data flow criteria will be defined for such coupling variables.

NPTEL

NPTEL

Moving on what are we going to do in the next lecture in next lecture will test will look at how to test for these problematic features such that we detect the faults that we understood in this lecture.

If you remember when we did integration testing data flow using graphs and all that we did data flow testing and we introduced the notion of coupling variable. We will extend it to object oriented coupling and we will define data flow criteria over these coupling variables. So, that is it for today.

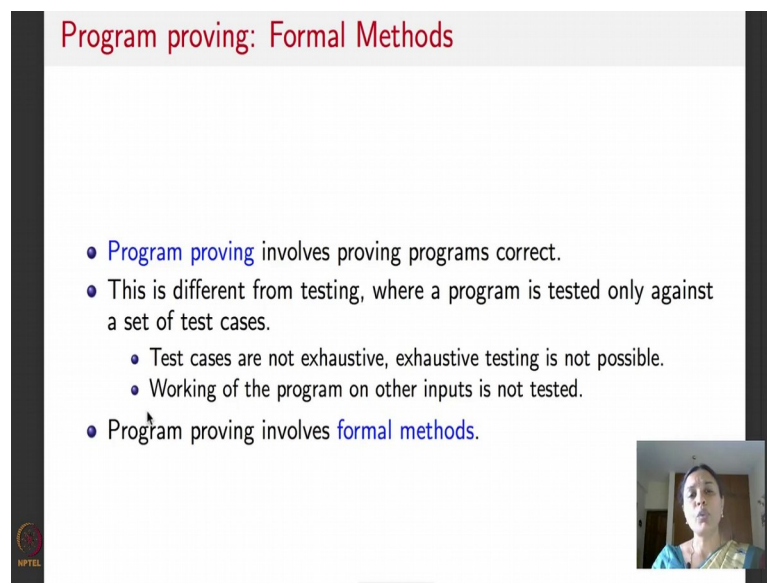
Thank you.

Software testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 50
Symbolic Testing

Hello, there I am going to begin a completely new module with this lecture. What we are going to do a bag of techniques called symbolic testing techniques which I said I would do at the end of this course. So, symbolic testing broadly includes a new set of testing techniques, which are actually speaking more than a decade, more than 3 decade old, but have become popular now. So, over the next 4, 5 lectures we will exhaustively understand symbolic testing.

(Refer Slide Time: 00:43)



Program proving: Formal Methods

- Program proving involves proving programs correct.
- This is different from testing, where a program is tested only against a set of test cases.
 - Test cases are not exhaustive, exhaustive testing is not possible.
 - Working of the program on other inputs is not tested.
- Program proving involves formal methods.

NPTEL

Video inset showing Prof. Meenakshi D'Souza speaking.

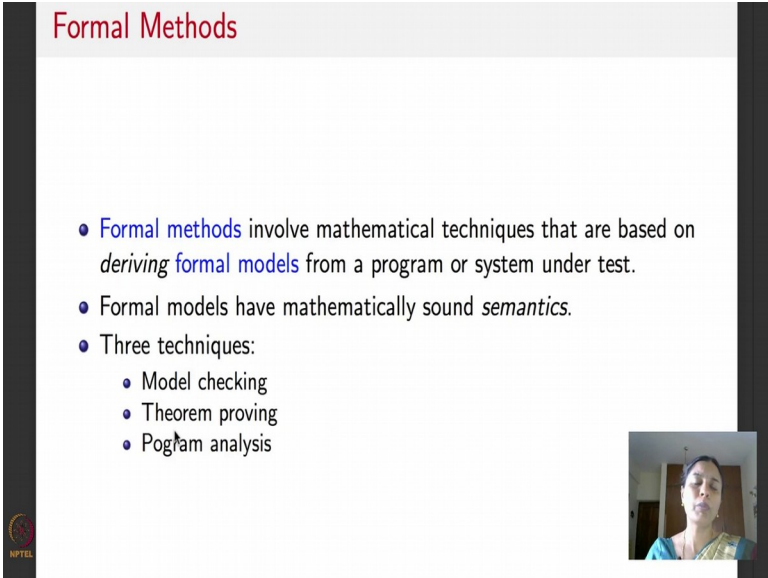
So, before I move on with symbolic testing if you remember right in the first week, I told you about the things that we will not do as the part of the testing course, and one of the things that I said we will not do is related to proving programs correct, which broadly use a bag of techniques collectively known as formal methods. Program proving means taking a program and showing that it works correctly no matter what. What do we mean by no matter what? That is if given any input that is legally acceptable by a program, it will always produce correct output. Now one thing to be noted before we move on is that program proving is different from program testing. When I test a program all that I do is

to give some inputs using carefully designed tests selection methods which we saw almost throughout the course, and check if the program works on these test inputs correctly.

Now, we all know exhaustively you cannot go on giving test inputs to a program, it is not feasible and it is not necessary also. That is not the goal for testing. It is to be able to cleverly design test case techniques to find the error. Given that we cannot exhaustively test a program what we can do using testing is to show that on the inputs that we have tested, the program does the way it is supposed to be doing, it either works correctly or it has an error. We cannot make any claim about the inputs that we have not tested the program on. Hence program can never be proved to be correct on the other inputs.

Program proving deals with proving a program to be correct and that involves a bag of techniques as I told you which are called formal methods. What are formal methods? I am not going to introduce you even to a cursory extent of what formal methods is, I am just listing here so that if you are interested you could look up for other courses on NPTEL or other material in this area.

(Refer Slide Time: 02:30)




Formal Methods

- Formal methods involve mathematical techniques that are based on deriving formal models from a program or system under test.
- Formal models have mathematically sound semantics.
- Three techniques:
 - Model checking
 - Theorem proving
 - Program analysis

Formal methods is the word formal says involves mathematical techniques that are based on taking a particular software artifact co design requirements, you derive formal models from that and then you verify or validate a program. Formal methods because their formal, have mathematically sounds semantics. There are 3 broad categories of formal

methods available; model checking, theorem proving and program analysis. There is a currently running NPTEL course on model checking.

(Refer Slide Time: 03:18)



Program testing vs. proving

- Program testing and program proving can be considered as extreme alternatives.
- In testing, one can be assured that sample test runs work correctly by carefully checking the results.
- Correct execution of program for inputs not in the sample is still in doubt.
- In program proving, one formally proves that program meets its specification (of correct behavior) without executing the program at all.
- Not all proofs can be automated and most proofs need to be done by hand.

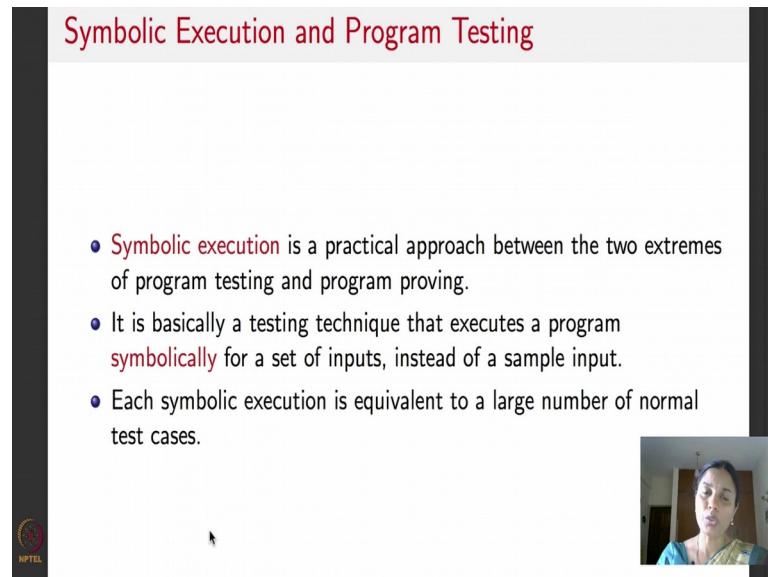
Now, program testing as I told you that is what we have dealt with in the course, testing a program. The other extreme is program proving which is proving a program correct. They can be thought of as extreme alternatives, one does not replace the other. They both can be used together and it does not if you have proved a program correct that does not mean that you need not test a program, if you have tested a program it does not mean that you need not prove a program to be corrected. They are complementary in fact, they consider extreme alternatives.

In testing as I told you can be assured that the sample test runs work correctly by checking the results, the actual output does not match the expected output or not. And correct execution of program on inputs that have not been tested is still in doubt. No sweeping claims about the program is correct, because it is working on all my test inputs that is wrong to state. But program proving you do not actually execute it one vector at a time one test vector at a time, you use a bag of techniques that are listed here to mathematically prove that a program meets its specification.

What is its specification? Is any requirement about a property of a program; could be a requirement that states that the program needs its functionality of a certain kind and so on. It may or may not involve executing the program and the thing to be noted is that

unlike testing, where there is heavy automation, program proving cannot be automated at all. Typically model checking is where there is automation possible, but it is not scalable for all kinds of software. In theorem proving and in program analysis full automation is never possible, proofs are done by hand.

(Refer Slide Time: 05:00)



The slide has a title bar at the top that reads "Symbolic Execution and Program Testing". Below the title, there is a list of three bullet points:

- Symbolic execution is a practical approach between the two extremes of program testing and program proving.
- It is basically a testing technique that executes a program symbolically for a set of inputs, instead of a sample input.
- Each symbolic execution is equivalent to a large number of normal test cases.

In the bottom right corner of the slide, there is a small rectangular video inset showing a woman with dark hair, wearing a blue and yellow patterned top, speaking.


What we are going to do symbolic execution as I told you. Symbolic execution can be thought of, or has been introduced by several papers that deal with this, as a practical approach that or else stands in between program testing and program proving. Symbolic execution is basically a testing technique, that is why we are looking at it in this course. It again executes a program, but it does not execute the program the way we have seen it till now, it executes the program in a particular style called symbolic execution that is the term. So, we will understand what that term is today. Each symbolic execution can be thought of as executing a program on large set of actual inputs.

(Refer Slide Time: 05:40)

Example 1

Consider the following program fragment:

```
1 Sum(a,b,c)
2   x = a+b ;
3   y = b+c;
4   z = x+y-b;
5   return(z);
6 end
```



So, I will explain it through an example, here is a small piece of code very small piece of code what does it do? It takes 3 numbers a, b and c. Do not worry about what the types are. It is not given here, it is not important for us. Let us assume without loss of generality that they are integers, takes these 3 numbers a b and c and then what does it do it computes the sum of a b and c. But it does it in the roundabout way, first it adds a and b stores it in x, then it adds b and c stores it in y.

Please remember it is supposed to add a with b and then with c, but by then what it does is that because it is adding b twice between x and y when it finally, computes z by doing $x + y$ it subtracts one b. This is just some way of writing program may not be the most intuited way, but for some reason let us say it takes it like this, and then it returns z which is the sum of a b and c. Let us say I give you the task of testing this program, what will you do? You will give some values to a, b and c based on some criteria and then test the program. You will see what for these values of a, b and c that you have given, what is the expected output. Then you will run this piece of program and record the actual output and see if the expected output matches the actual output. If it does then you say that the test case is passed.

(Refer Slide Time: 07:02)

Normal execution of Sum on inputs

Normal execution of program Sum on inputs 1, 3, 5 is given in the table below:

After stmt.	x	y	z	a	b	c
1	?	?	?	1	3	5
2	4	-	-	-	-	-
3	-	8	-	-	-	-
4	-	-	9	-	-	-
5	returns 9					

In the above table, - represents unchanged values and ? represents undefined (uninitialized) values.

So, our actual test statement will look like this. So, let us say the inputs you are going to give to the program a, b and c are 1, 2 and 3. I have written it here: a is 1, b is 2, c is 3. I will come back and tell you what this question marks are. What is question marks mean is that as you give statement, as you give inputs a, b and c as 1, 2 3, let us say you maintaining this table which records what happens after these 5 statements in the program are executed. What are the 5 statements in the program? You go back to the program, these are the 5 statements. The first statement is just the function name second statement is this, third statement is this, fourth statement is this, fifth is the return statement, sixth is the end statement.

So, you keep this table, where you give inputs a, b and c as some concrete values, in this case let us say 1, 2 and 3, and then in the statement you record the values of the variables in the program as the program executes. There are 3 other variables in the program x, y and z. x and y can be thought of as internal variables, z is the variable that is returned. So, after statement one which is the first statement in the program, which let us you pass inputs as a, b and c, x, y and z are not assigned any values, that is why there are these 3 question marks. After statement number 2 executes x becomes a + b, what is a, 1. What is b? 3. So, a + b is 4. So, x becomes 4.


I have put these dashes they can be interpreted as values have not changed, in the sense that I still do not know what y is, I still do not know what z is, a remains 1, b remains 3, c

remains 5. If this is confusing you copy exactly what was there in the previous row here that is what we mean by this dash. Now after statement number 2 which computes y as $b + c$, y gets assigned value 8 all other values remain the same. Similarly after statement number 3, what does z do? z is $x + y - b$. That will be $4 + 8$ which is 12, b is -3, $12 - 3$ is 9. What is line number 5 do? Line number 5 returns 9. Using this statement you see actual output is the same as expected output, yes, then you say my test case is passed, otherwise you say my test case is failed.

Now, let us say you decide to give another set of inputs to the program Instead of 1 3 and 5, may be you will give 0, 0 and 0 to see what happens how the program behaves. For the same set of inputs, you will again do the same thing, you will give the inputs look at the expected output look at the actual output compare and see if the test case is passed or failed. You can repeat the process for another set of inputs, repeat the process for another set of inputs and do on carry on till test a criterion is met, or till you are satisfied as a tester that the program is working correctly or when you reach a state where you know that the program is erroneous. In this case the program segment that is given here is not erroneous. So, it does work correctly.

(Refer Slide Time: 10:23)

Symbolic execution of Sum



After stmt.	x	y	z	a	b	c	PC
1	?	?	?	α_1	α_2	α_3	true
2	$\alpha_1 + \alpha_2$	—	—	—	—	—	—
3	—	$\alpha_2 + \alpha_3$	—	—	—	—	—
4	—	—	$\alpha_1 + \alpha_2 + \alpha_3$	—	—	—	—
5	Returns $\alpha_1 + \alpha_2 + \alpha_3$						

The above table represents symbolic execution of $\text{Sum}(\alpha_1, \alpha_2, \alpha_3)$. Path Condition/Constraint is abbreviated as PC.

Now, let us assume instead of giving these concrete values as 1, 3 and 5, or any other concrete values let us say I want to execute it symbolically. What I will do in that case is that, I will say a is not 1 or any concrete value, a is some symbolic value, let us say α_1 . b

is another symbolic value let us say α_2 , c is another symbolic value let us say α_3 . I will come back and tell you what this last column is. The rest of the table exactly is the replica of this table, but instead of having concrete values here, it executes it on symbolic values. How do you interpret symbolic values at this stage?

Think of symbolic vales as being place holders for concrete values. So, when I say a is α_1 , you can think of a as being taken the symbolic value α_1 , where α_1 will eventually be 1 concrete value. Similarly b is also symbolically α_2 , α_2 could represent any abstract value for b that matches the type of b, eventually be substituted by a concrete value for b. Similarly for c. The rest of the table is the same--- it says in statement 1, x, y and z are not yet assigned. After statement 2 x becomes $\alpha_1 + \alpha_2$ y and z remain the same, all these remain the same. x becomes, x remains the same here, after statement 3 y becomes $\alpha_2 + \alpha_3$ z is not yet assigned, α_1 α_2 α_3 are the same, after statement 4, x and y are the same as before, z becomes this and so on is this clear? What does the program return?

This table is the same as this table there are 2 differences; one is here a, b and c are given concrete values as input, a, b and c are given abstract or symbolic values as input everything else is the same. The last column represents what is called a path constraint or a path condition. PC is short form for path constraint. Path constraint here is marked as true. You can think of it as you remember our reachability, infection and propagation condition, you can think of PC as a set of conditions that collect those predicates that predicates, corresponding to reachability, infection and propagation.

If you see this program, it does not have any decision statements any branching statements. So, reachability is always true. Infection and propagation will also be eventually true. There is nothing like path constraint here. So, the path constraint is marked as true right. So, this is how a program is symbolically executed. So, just to repeat what I said, you want to symbolically test or execute a program, the program looks like any other normal program written let us say C or Java or whatever programming language. For normal testing you would give concrete values to inputs like in this case you would give a, 1, b, 3, c, 5. In symbolic testing you would give symbolic values to inputs.

And then what do you do? You collect expressions corresponding to internal variables because you have not given concrete values you cannot evaluate an expression $x = a + b$, you can only collect another expression in terms of the symbolic variables. One natural doubt that might come to your mind or that you might ask your self is, what am I doing differently than what is given in this program here? Instead of writing $a + b$, I am writing $\alpha_1 + \alpha_2$, instead of writing $b + c$ I am writing $\alpha_2 + \alpha_3$ and so on. That is natural, in some sense we are not doing anything. We just saying that α is an arbitrary place holder value for a β is another arbitrary place holder value for b, α , sorry α_2 is another arbitrary place holder value for b, and α_3 is an arbitrary place holder value for c, that is exactly what is symbolic execution.


(Refer Slide Time: 14:25)

Example 2

Consider the following program:

```

1 int twice (int v) {
2     return 2 * v;
3 }
4 void testme (int x, int y) {
5     z = twice(y);
6     if(z == x) {
7         if(x > y + 10)
8             ERROR;
9     }
10 }
11 int main() {
12     x = sym.input();
13     y = sym.input();
14     testme(x, y);
15     return(0);
16 }
```



I will illustrate it with another example, this time it is a slightly bigger code. So, let us go through the code first. So, here is a program. So, it has a function called twice which takes an argument of type integer given as variable v and it returns an integer type. It is just a one line function. What does it do? It multiplies v by 2 and returns it nothing else. Now there is another program which is called test me. These names are given just for improving readability there is nothing else to it. This program test me returns what? It takes 2 arguments x and y.

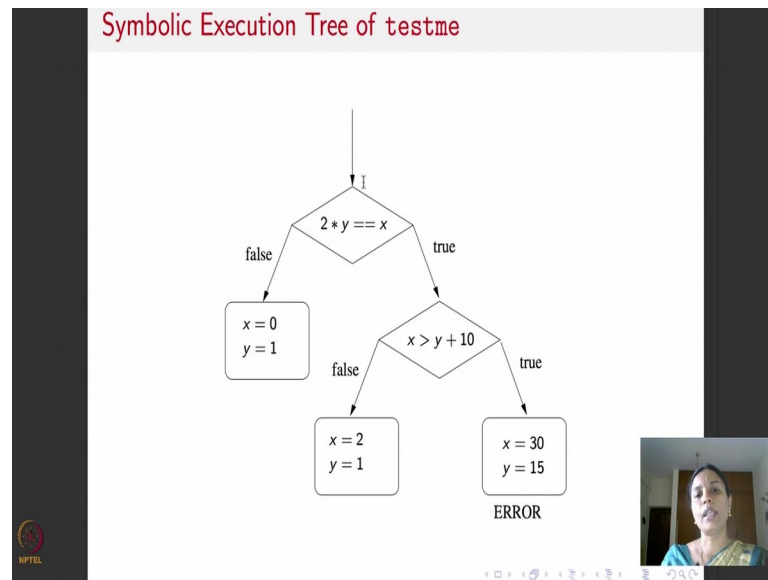
So, what does it do the first line it calls this function twice. To it might as well directly said $z = 2y$, but I have just said this to illustrate a particular point that I will come back

to. So, it calls the function twice which basically multiplies its argument by 2 assigns it to z and then what does it do. It says if z is equal to the other input x , then you go and check for one more condition. If $x > y + 10$, then you say the program reaches an error statement reaches an error state or error condition. I have not explicitly written what the error condition is, but indicated more like that there is an error just by writing it as error. It may not syntactically be legal in any programming language, but think of it as representing an abstract piece of code that indicates an error or you can think of it as printing an error state.

So, what does this program do? This program is very simple: takes 2 arguments x and y does z is equal to $2y$, then passes 2 if conditions, checks, first checks if $z = x$, then checks if $x > y + 10$. Both the if conditions have passed it says there is an error, otherwise it does not do anything, this is the program. Then this is being called by main calls tests me, and then it says return 0 and I have also said what are inputs x and y to test me? I say x is given as a symbolic input read this `sym_input` as x is being given as a symbolic input, similarly y is being given as a symbolic input.

Now, before we move on and understand how symbolic execution works for this program, what do I hope to achieve? I am telling you that there is an error in the program using symbolic testing my goal would be to find this error. When can I reach this error? I can reach this error if this if condition passes and if this if condition passes. You might want to spend a minute thinking about which will be the values of x and y for which both the if conditions will pass. By this is a small enough program if I look at this program for a couple of minutes and study its code, I will be able to come up with 2 values for x and y such that both the if conditions pass and I will reach the error statement for sure.

(Refer Slide Time: 17:35)



Let us say instead of staring at this program and thinking about it manually yourself, you want to be able to systematically do it. How will you do it? Let us say here is a way. So, I will draw what is called an execution tree of a program. What is this execution tree of a program tell you? This can again be related to our reachability, infection and propagation condition. Right in the very first statement after calling this function twice(y), I can always reach here. There are no if conditions, this is the first place where a decision takes place. If this if condition itself is false then this program exits.

Suppose this if condition is true then it goes into the second if condition, that is what this node captures here. So, it says, this is a first if statement that is encountered, it can be true or it can be false. If it is false then the program does something, if it is true then the second if statement is tested which in turn, can be true and false. Now what are these 3 leaf nodes in the tree that I have given you. I have given you an example test case that will make this if condition false. For example, if x is 0 and y is 1, then this if condition will be false why? Because it says 2 is equal to 0 which is false.


So, this can be thought of as a test case that will make this false. If this is true which in some test case it could be either this or this, I will go ahead and test the next statement. So, a test case for example, x =2, y = 1 will make this one true and then this one false. Similarly a test case for example, x = 30 and y = 15, will make both the if statements true and this is the test case that corresponds to the error.

So, ideally to exhaustively test this program or to do edge coverage or branch coverage or predicate coverage for this program, you make this statement true once false once, you make this statement true once false once, that is what this does. This test case will make this statement false, the first if, then any of these test cases will make the if, this if statement true and this exclusively make this second if statement false, this will make the second if statement true which will hit the error. So, symbolic execution, what it does is, it will go back to this program, it try to not give concrete values for x and y, it will give symbolic values for x and y, and what it will do it collect constraints corresponding to these if conditions. Towards the end, it will try to solve the constraint and try to automatically obtain these value, that is what symbolic execution does.

(Refer Slide Time: 20:11)

Symbolic Execution

- Symbolic execution uses **symbolic values** instead of concrete data values as input.¹
- Program variables are represented as **symbolic expressions** over the symbolic input values.
- Hence, output values computed by a program are expressed as a function of symbolic input values.
- **Symbolic state** σ maps variables to symbolic expressions.
- **Symbolic path constraint**, PC, is a quantifier-free, first-order over symbolic expressions.



So, symbolic execution uses symbolic values instead of concrete values as inputs, inputs means test case inputs. Program variables are represented as symbolic expressions over the symbolic input values. Like for example, here going back to the previous sum code, there were 3 program variables x, y and z, when I symbolically executed them, I represented them using symbolic expressions I represented x as $\alpha_1 + \alpha_2$, y as $\alpha_2 + \alpha_3$ and so on right. So, that is what is written here. Program variables which are internal variables in a program that do not correspond to inputs now become symbolic expressions as the program runs.


The symbolic expression can be thought of first it is an arithmetic expression that is there already in the program, but has symbolic variables because it is not being evaluated. Output values are also a function of symbolic input variables. What is a symbolic state? Symbolic state at any point in a time gives me the symbolic values of all the variables. Like for example, here going back to the sum program, if I read this at any point in time I get the symbolic state. It says x is $\alpha_1 + \alpha_2$, y is $\alpha_2 + \alpha_3$, z is not yet assigned value, a is α_1 , b is α_2 , c is α_3 , the program is executing statement number 3. That is what is symbolic state will say.

Symbolic path constraint, it is what? We have introduced logic, if you remember as per the logic that we introduced, is a quantifier free predicate logic formula or it is a quantifier free first order logic formula. Predicate logic is the same as first order logic. Obviously, there are no quantifiers and the path constraint is a normal predicate logic formula. If you go back here, the path constraint to reach this statement will be this is true and this is true. The path constraint to reach this statement would be not of this because that is when this becomes false. For an example program like this, there are no path constraints or path constraint is just the Boolean constant true, because there are no decision statements in this program.


So, I will just quickly repeat what I said in this slide. Symbolic executions symbolically executes the program over inputs, all the internal variables of program, every internal variable will have some expression in the program that evaluates it. Instead of evaluating it you just substitute it with symbolic variables and keep it as symbolic expressions. A symbolic state will be a large to pull that gives symbolic values or symbolic expressions for the inputs and the variables in the program. Along with that, I collect path constraints which can be thought of as, all the decision statements that you encounter in the program take the statement as it is when it is true, takes the negation of the statement when it is false keep AND-ing them and since there are no quantifiers here, it is a quantifier free first order logic or a predicate logic formula.

(Refer Slide Time: 23:25)

Symbolic execution in software testing




- In software testing, symbolic execution is used to generate a test input for each execution path of a program.
- An execution path is a sequence of true and false, where true (respectively, false) at the i th position in the sequence denotes that the i th conditional statement encountered along the execution path took the "then" (respectively, "else") branch.
- All the execution paths of a program can be represented using a tree, called the **execution tree**.



What is symbolic execution do? In software testing symbolic execution is used to generate a test input for each execution path of a program. As I told you if you go back to this tree, an execution path is a sequence of true false values right? If I say this is a path of length one which makes first condition false, this is the path of length 2 which makes the first condition true, second condition false and so on. So, a path constraint or an execution path is a sequence of true and false values, where true at the i th position in the sequence denotes that the i th conditional statement is true or it took the then branch and a false in the i th position denotes that the i th conditional statement took the else branch. This should be obvious as we see it in that example and we represent the execution path using an execution tree, this is an execution tree.

(Refer Slide Time: 24:24)

Symbolic Execution: Symbolic state σ



- At the beginning of symbolic execution: σ is initialized to an empty map.
- At a read statement: symbolic execution adds the mapping assigning the variable being read to its new symbolic variable, to σ .
- At every assignment $v = e$: symbolic execution updates σ by mapping v to $\sigma(e)$, the symbolic expression obtained by evaluating e in the current symbolic state.

So, at the beginning of the symbolic execution the state, symbolic state sigma is assigned an empty map, which is dash dash dash, a question mark question mark question mark. At every read statement assigns a symbolic variable, at every assignment statement it assigns a symbolic expression, like here. For example, there was this statement which can be thought of as a read statement, after statement one symbolic values are assigned, after statement 2 symbolic expressions is assigned, that is what I mean. At every read statement, symbolic execution adds the mapping assigning variable being read to it is symbolic variable. At every assignment statement it adds symbolic expression.

(Refer Slide Time: 25:06)

Symbolic execution: Path constraint PC

- At the beginning of symbolic execution: PC is initialized to true.
- At a conditional statement $\text{if}(e) \text{ then } S_1 \text{ else } S_2$: PC is updated to $\text{PC} \wedge \sigma(e)$ ("then" branch) and a fresh path constraint PC' is created and initialized to $\text{PC} \wedge \neg \sigma(e)$ ("else" branch).
- If PC (PC') is satisfiable for some assignment of concrete to symbolic values, then symbolic execution continues along the "then" ("else") branch with σ and PC (PC').
- Note: Unlike concrete execution, both the branches can be taken resulting in two execution paths.
- If any of PC or PC' is not satisfiable, symbolic execution terminates along the corresponding path.

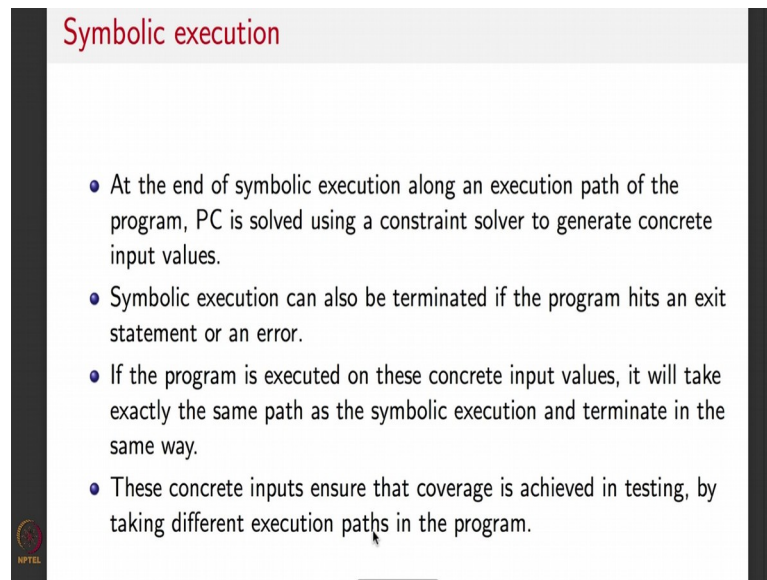
Now, path constraint, path constraint initially is true to start with. The moment you get a first if statement if e then S else 1, S1 else S 2, this is how it looks like. e is actually the condition whatever your path constraint was you and it with sigma of e which means take e substitute with symbolic expression AND it. This is for the then branch when that conditional value is to true and you have to and it with negation of sigma of e to represent the else branch for the condition to be false.

Like for example, if I go back here there were 2 path constraints, one for this if and one for this if. Till I come here path constraint is true and then I will do $\text{true} \wedge x \text{ for } 1$, and I will do $\text{true} \wedge \neg z \text{ is equal to } x$. So, $\text{true} \wedge z \text{ is equal to } x$ will make this takes to true branch true, and $\neg z \text{ is equal to } x$ will make this statement take the false branch. And when I come here I already had $\text{true} \wedge z \text{ is equal to } x$, if I negated with AND of if I and it with $x > y + 10$ then I get this if I and it with $\neg (x > y + 10)$ then I get this. So, that is what I said. As you are running a program executing a program you would have a path constraint, for every if statement that you encounter you AND it with e once, and you end it with negation of e once. AND-ing it with e intuitively represents the program taking the then branch and ending it with negation of e represents the program taking an else branch.

Finally what you do you would have collected a path constraint like here you would have collected this constraint and this constraint. You solve it, means you give it to a solver will return value say for this to be true and this to be true here is one value. If x is 30 y is 15 both are true. This becomes your test case that you execute the program on and similarly to how do you get this test value, you solve this and negation of this, then this will be the test case how do you get this test case you solve negation of this.

So, each path constraint is solved and if there is a assignment then the program will continue along the then or the else branch right. If a path particular path constraint is not satisfiable for some reason cannot be solved then you will say symbolic execution terminates then and there.

(Refer Slide Time: 27:39)



The slide is titled "Symbolic execution" in red text at the top. It contains a bulleted list of four points. The first point states that at the end of symbolic execution, a path constraint (PC) is solved using a constraint solver to generate concrete input values. The second point states that symbolic execution can be terminated if the program hits an exit statement or an error. The third point states that if the program is executed on these concrete input values, it will take exactly the same path as the symbolic execution and terminate in the same way. The fourth point states that these concrete inputs ensure that coverage is achieved in testing by taking different execution paths in the program. There is a small NPTEL logo in the bottom left corner of the slide.

- At the end of symbolic execution along an execution path of the program, PC is solved using a constraint solver to generate concrete input values.
- Symbolic execution can also be terminated if the program hits an exit statement or an error.
- If the program is executed on these concrete input values, it will take exactly the same path as the symbolic execution and terminate in the same way.
- These concrete inputs ensure that coverage is achieved in testing, by taking different execution paths in the program.

So, at the end of symbolic execution path constraint is solved using a constraint solver. There could be third party constraint solvers. If you remember when I introduced propositional logic to you I told you that the satisfiability problem for propositional logic is an NP-complete problem, it is a hard problem. And the satisfiability problem for predicate logic in general is undecidable, but there are lots of constraints solvers: dReal is one of them that I use, that you can use to solve these path constraints.

And if concrete test values come, then they become your concrete test inputs. What do these concrete test inputs achieve for you? Let us say the concrete test inputs will achieve, by how many path constraints you have here? 3 different path constraints. Two path constraints, but their combinations give you 3 conditions. By giving it in solving what you guaranteed is, you are guaranteed exhaustive coverage for program. With just 3 test cases, you can completely cover this program. Test this program for every behaviour that is possible in the program that is what is written here. So, it says that the program is executed on the concrete input values that the constraint solver gives you, it will take exactly the same path as the symbolic execution and terminate in the same way. These concrete inputs ensure that coverage is achieved in testing. You will be able to take different test paths in the program.

(Refer Slide Time: 29:09)

Example 2, re-cap

Consider the program in example 2 again:

```
1 int twice (int v) {  
2     return 2 * v;  
3 }  
4 void testme (int x, int y) {  
5     z = twice(y);  
6     if(z == x) {  
7         if(x > y + 10)  
8             ERROR;  
9     }  
10 }  
11 int main() {  
12     x = sym_input();  
13     y = sym_input();
```

We will see a few more examples where this is illustrated very clearly. Like for example, this is that we saw the same code I am showing you here again.

(Refer Slide Time: 29:17)

Symbolic Execution: Example 2

- After executing statements 12 and 13: $\sigma = \{x \mapsto x_0, y \mapsto y_0\}$, where x_0 and y_0 are two initially unconstrained symbolic values.
- After executing line 5: $\sigma = \{x \mapsto x_0, y \mapsto y_0, z \mapsto 2y_0\}$.
- After line 6: two instances of symbolic execution are created with path constraints $x_0 = 2y_0$ and $x_0 \neq 2y_0$.
- Similarly after line 7: two instances of symbolic execution are created with $(x_0 = 2y_0) \wedge (x_0 > y_0 + 10)$ and $(x_0 = 2y_0) \wedge (x_0 \leq y_0 + 10)$.
- Solving the path constraints, we get three instances of symbolic executions resulting in concrete test inputs $\{x = 0, y = 1\}$, $\{x = 2, y = 1\}$ and $\{x = 30, y = 15\}$.

I will just walk you through the steps of symbolic execution, after executing these statements 12 and 13, the symbolic inputs x_0 and y_0 are assigned to x and y and after executing line 5 which is this line, x_0 stays as x_0 , y_0 stays as y_0 , z becomes $2y_0$. This is a symbolic expression and after line 6 where you encounter first if statement, one which passes x as $2y_0$, the other one is failing x as equal to $2y_0$.

Now, you in an passes will go to the inner if statement where you take this if condition $x_0 = 2 y_0$, and AND it with the condition of the second if statement and you take the same condition $x_0 = 2 y_0$ and AND it with the negation of the if statement which amounts to $x_0 \leq y_0 + 10$. So, you have 3 constraints that you have collected, first one is $x_0 \neq 2 y_0$, second one is $x_0 = 2 y_0 \wedge x_0 > y_0 + 10$, and third one is this.

You let us say you give these 3 constraints to a constraint solver, then you will get some values like this. These are just one set of sample values, any values that solve these equations satisfy these equations are good enough, this is a small example. So, you could just look at it and do this yourself. So, the value of symbolic execution may not be seen, but let us say you are running it on a large enough example, then actually collecting these constraints and solving them will help you to drive program execution to a particular path.

So, symbolic execution is valuable from that point of view. So, I will stop here. In next lecture I will continue with symbolic execution. We will look at additional examples of code that involves loops and so on.

Thank you.

Software testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore



Lecture – 51
Symbolic Testing

Hello again, I will continue with the symbolic execution. If you remember in the last class, I had introduced symbolic execution to you shown you a couple of examples and given you an overview of the technique. What I am going to do now is to continue with it, we look at the third example.

(Refer Slide Time: 00:20)

Symbolic Execution: Example 2

- After executing statements 12 and 13: $\sigma = \{x \mapsto x_0, y \mapsto y_0\}$, where x_0 and y_0 are two initially unconstrained symbolic values.
- After executing line 5: $\sigma = \{x \mapsto x_0, y \mapsto y_0, z \mapsto 2y_0\}$.
- After line 6: two instances of symbolic execution are created with path constraints $x_0 = 2y_0$ and $x_0 \neq 2y_0$.
- Similarly after line 7: two instances of symbolic execution are created with $(x_0 = 2y_0) \wedge (x_0 > y_0 + 10)$ and $(x_0 = 2y_0) \wedge (x_0 \leq y_0 + 10)$.
- Solving the path constraints, we get three instances of symbolic executions resulting in concrete test inputs $\{x = 0, y = 1\}$, $\{x = 2, y = 1\}$ and $\{x = 30, y = 15\}$.





So, here is another program; this time this program has a loop and the idea is to understand how symbolic execution works on programs that have loops.

(Refer Slide Time: 00:28)

Example with loops: Example 3

```
1 void testme_inf() {
2     int sum = 0;
3     int N = sym_input();
4     while (N > 0) {
5         sum = sum + N;
6         N = sym_input();
7     }
8 }
```



So, here is a piece of code is named as test me infinite, inf short for infinite, do not worry too much about the name; let us understand what the program does. It has an internal variable called sum which is an integer type initially assigned to 0, then it takes as another variable which is of type integer called N. How does it take it as? It takes it as input, it takes it as symbolic input which means what you take it as a symbolic input eventually substitute it with concrete value and then the program has piece of code.

What does that code do? It has a while loop which says as long as the input that you have read, of integer type, is an input greater than 0, you keep adding it to the sum that you have collected till now and take the next input. If you happened to get an input that is less than or equal to 0, then this while loop gets aborted. Otherwise, this program, this while loop will run for ever as long as somebody gives it positive inputs. So, that is why this is called infinite. So, this program has a problem. The problem is, this program, the loop in the program, could be non-terminating and the whether the loop terminates or not is fully based on the next input that the user gives. I want to show or flag this as a potential error in the program because it has an input statement inside the while loop, it can go on running as long as the condition of the while loop passes.

What is the condition of the while loop? Condition of the while loop just says that the input that you have read just now is it greater than 0 or not. So, this program will go on running as long as somebody gives it positive inputs. I want to use symbolic testing to

identify this as a problem with the program and tell the user that there is a problem. How will I do that? If you remember the steps of symbolic testing, what are the steps? Instead of giving concrete values to inputs give symbolic values for every internal variable, collect a symbolic expression which is basically substitute the value of the input that comes in the expression with its symbolic value, collect path constraints that will tell you about the decision statements in the program. If the decision statements were simple if statements like the ones we saw earlier there would have been no problems. Here the decision statement, in this program is a while statement and it is a while statement which says $N > 0$.

Now, you could go by the previous times example and say that the path constraint will be symbolic value of $N > 0$, but then just that as the path constraint does not tell you; when this while loop is going to stop. You also remember, you also have to give a negation of the path constraint. For every path constraint that you encounter is the decision statement, an if or a while, you have to be able to give its negation also. What is negation of $N > 0$? That is $N \leq 0$ and how will I give it as a path constraint for this, because it is a while loop it has to run for one or more iterations. In fact, it runs as long as $N > 0$ and when N is less than or equal to 0 it will stop running. Is that clear, that is what I specify in the path constraint.


This is what I was trying to explain it to you, I will go through this slide.


(Refer Slide Time: 04:05)

Symbolic execution with loops

- The loop in example 3 has an infinite number of execution where each execution path is either a sequence of arbitrary number of true-s followed by a false or a sequence of infinite number of true-s.
- PC for the loop with a sequence of n true-s followed by a false is
$$(\bigwedge_{i \in [1, n]} N_i > 0) \wedge (N_{n+1} \leq 0)$$

where each N_i is a fresh symbolic value.
- σ at the end of the execution is $\{N \mapsto N_{n+1}, \text{sum} \mapsto \sum_{i \in [1, n]} N_i\}$.
- In general, symbolic execution of code containing loops or recursion may result in an infinite number of paths if the termination condition for the loop or recursion is symbolic.

 In practice, one needs to put a limit on the search, e.g. a timeout, or a limit on the number of paths, loop iterations or exploration depth.



Now, the loop in the example that we are seeing in this slide, example 3 has infinite number of execution paths as I told you where each execution path is a sequence of arbitrary number of true-s followed by a false, when it stops, or it can go on as a sequence of infinite number of true-s. So, the path constraint for a loop with the sequence of N true-s where N is unknown, it is completely dependent, n is completely dependent on the user. This while loop let us say the user tries to give positive values of N first 3 times fourth input that he gives is negative, the while loop will stop.

Let us say the user tries to give positive values of N 100 times, 100 and first input this gives is negative then, 100 and after 100 iterations the while loop will stop. User could decide to give first 10,000 values as positive and then 10,001 th value as negative, the while loop will run till then. User continuously gives positive values, while loop will go on running that what is captured here. It says that there is some N whose value, I do not know and the path constraint says till that value of N that is for I is equal to one to small n , all the numbers that I get capital N i's are greater than 0 and after this N , there is a $n+1$ th value which is less than or equal to 0.

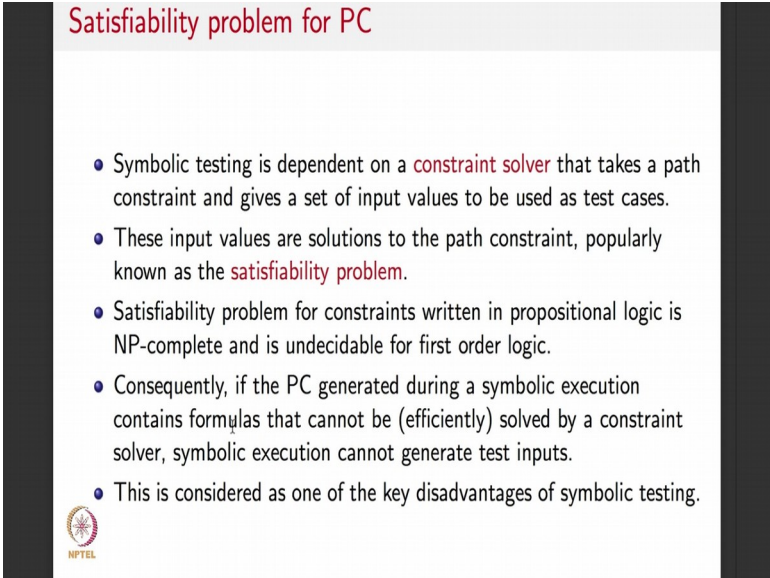
This N may or may not exist. If this then does not exist means what for no N , $N + 1 \leq 0$ means, for every N and I will be greater than 0 which means the while loop can go on executing. That is what captures that succinctly and what will happen after at each step of the execution? Sum gets assigned to sum plus that value of N i and the new input of N i is read that is what is represented here sigma which represented the symbolic state which I had explained in the previous lecture looks like this.

There are 2 variables in the program N and sum. N will stop when the program goes at the end of execution when the program gets $N + 1$ and what will sum be? Sum will be the summation of all the inputs that you have read till now. So, in general by using this example, what have we learnt? If I am doing a symbolic execution of a code that contains loops or recursion; recursion I have not shown you, but that holds for recursion also, it might result in an infinite number of the paths, if the termination condition for the loop is such that, it is also symbolic. For these kind of things, usually what people do is that they put a time out, they wait and then they say, no, there is a problem with the program.

I am doing this for too long, then they go and look at it and abort or they limit the number of paths, loop iterations or the exploration depth. So, this is how you collect symbolic constraints for loops. So, symbolic constraints for loop will have a particular number of iterations where the path constraint for the loop turns out to be true and then after the specified number of iterations the path constraint will be false.


In this case why I showed you this example directly is that in this case, the next iteration of the loop is dependent on the structure of the loop. This is particularly problematic case. So, I do not know whether I can stop it after a finite number of iterations or not. Or I do not know whether that n will exist or not. That is the case, then symbolic execution may or may not terminate and that truly reflects the problem in this program also, because concrete execution of this program also may or may not terminate. It will not terminate if a user continuously gives positive values. It will terminate, if the user eventually gives a 0 or a negative value.

(Refer Slide Time: 08:14)



Satisfiability problem for PC

- Symbolic testing is dependent on a **constraint solver** that takes a path constraint and gives a set of input values to be used as test cases.
- These input values are solutions to the path constraint, popularly known as the **satisfiability problem**.
- Satisfiability problem for constraints written in propositional logic is NP-complete and is undecidable for first order logic.
- Consequently, if the PC generated during a symbolic execution contains formulas that cannot be (efficiently) solved by a constraint solver, symbolic execution cannot generate test inputs.
- This is considered as one of the key disadvantages of symbolic testing.



So, now what is the biggest problem with the symbolic execution? Biggest problem with symbolic, when the symbolic execution end? When we are able to solve the path constraint and collect a set of test cases. In the case of this example, this was the path constraint to be solved. What is this path constraint solving means, find some N such that you give the first N values as numbers that are positive and $N + 1$ th value as numbers that is ≤ 0 . We can always find like this. As I told you the small n could be 100,

whatever it will be satisfied. What was the path constraint for this one? For this piece of program that was given here, right, 3 path constraints was there one first if statement negating to be false and the second one was first and second; if statement both were true first if statement was true, second if statement was false, all the cases these path constraints have to be automatically solved. These were small programs that I told you for illustrative purposes, but for large program, imagine a large program with 10000 lines of code several different if statements and you symbolically execute, you collect a large constraint.

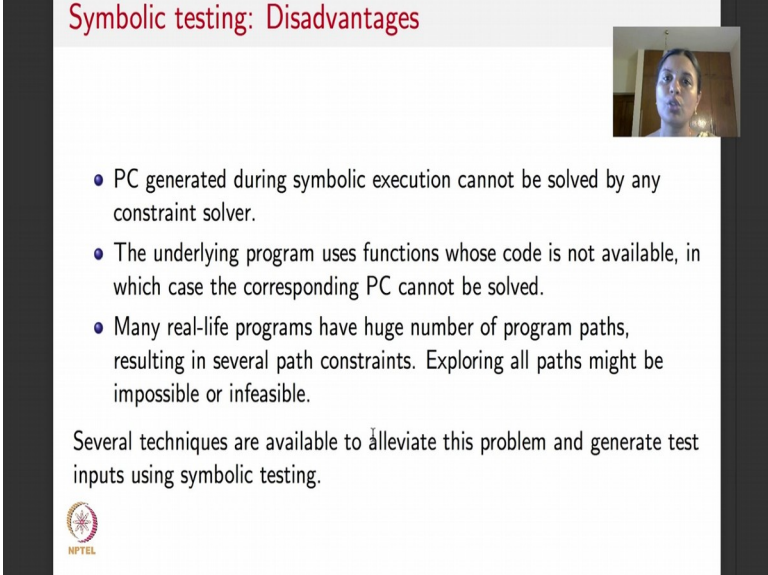
So, you have to give it to a constraint solver expect the constraint solver to be able to solve the path constraint for you. As I told you solving path constraints means looking at satisfiability problem in logic. If you remember in the lecture on logic that I did towards the first half of the course where before we began logic based testing, I had told you that satisfiability problem is a hard problem. Even for elementary logic like propositional logic, there no known polynomial time algorithm for solving satisfiability. Or in other words satisfiability problem for this is NP-Complete. We are not even talking about propositional logic. We are talking about predicate logic which means if there are variables that are of type integers that are not always Boolean. Satisfiability problem for predicate logic or first order logic is un-decidable which means there is no algorithm to do satisfiability.

But there are lots of so called constraint solvers which use exclusively designed heuristic based techniques to do satisfiability I will point you to references a couple of them towards the end of this course. So, suppose you get a path constraint and you are able to solve it using a constraint solver then well and good, you can do symbolic execution and drive the program towards the desired paths that you need or cover all the paths in the program. But let us say you are able to not solve the path constraint and get a collection of inputs then there is a problem symbolic execution may not be useful.

Why will you not be able to solve a path constraint, because there could be functions like $f(x) \leq 0.5$ and you may not have the code corresponding to $f(x)$. Or let us say in a programming language like C, if you include the math.h library, you can write things like if $\log(x) < 0.005$ then you do something. Things like log may not be efficiently computable by constraint solvers.

So, there could be several different reasons why arbitrary functions and predicates cannot be computed. So, if one such problem exists and constraints solver says I am not able to solve, then symbolic execution is not particularly feasible for that. In fact, this is considered to be one of the key disadvantages of using symbolic testing.


(Refer Slide Time: 11:47)



Symbolic testing: Disadvantages

- PC generated during symbolic execution cannot be solved by any constraint solver.
- The underlying program uses functions whose code is not available, in which case the corresponding PC cannot be solved.
- Many real-life programs have huge number of program paths, resulting in several path constraints. Exploring all paths might be impossible or infeasible.

Several techniques are available to alleviate this problem and generate test inputs using symbolic testing.

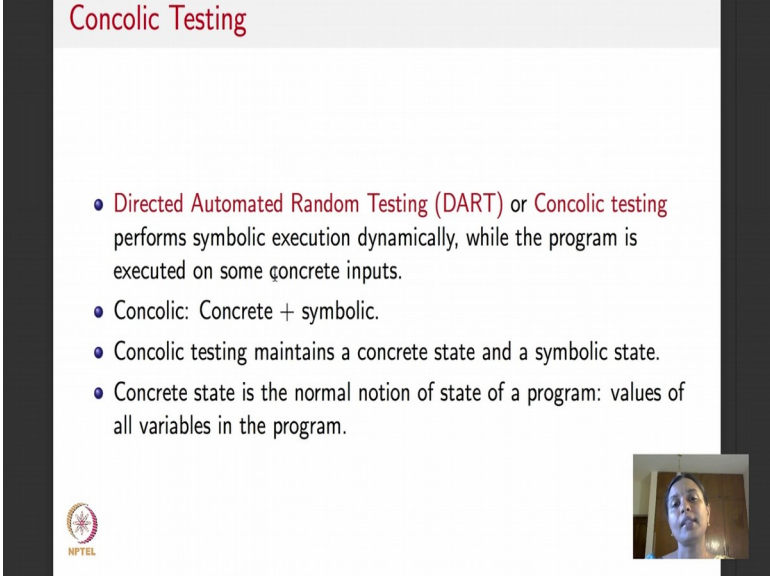


So, to summarize all the disadvantages of symbolic testing, the PC or the path constraint as I told you just now generated during symbolic execution may not be solvable by any constraint solver then you are stuck, you can never get test cases. Why will it not be solvable, because the underlying program could use a function whose code is not available. Then how will you know what $f(x)$ computes. This is very common when you are using web applications and all you might use functions whose code is not available. And as I told you many real life programs which have a few million lines of code have several different program paths.

So, as you go on symbolically executing a program there could be several decision statements and you might end up collecting a really long path constraint which is very difficult to solve. It is so long, it has so many variables that several constraint solvers might not find it feasible to be able to solve. In which case exploring all paths may not be feasible. In fact, symbolic execution was first introduced in the year 1976, quite old, almost 40 years from now. But then it is come into use only in the past 10 years it was dormant mainly because of these disadvantages.

Recently if people have devised a lot of techniques from 2005 onwards where some of the disadvantages that I have listed here can be overcome.

(Refer Slide Time: 13:12)



The slide is titled "Concolic Testing" in red text at the top. It contains a bulleted list of four points. The first point is "Directed Automated Random Testing (DART) or Concolic testing performs symbolic execution dynamically, while the program is executed on some concrete inputs." The second point is "Concolic: Concrete + symbolic." The third point is "Concolic testing maintains a concrete state and a symbolic state." The fourth point is "Concrete state is the normal notion of state of a program: values of all variables in the program." In the bottom left corner, there is a small circular logo with the text "NPTEL" below it. In the bottom right corner, there is a small video inset showing a person's face.

- Directed Automated Random Testing (DART) or Concolic testing performs symbolic execution dynamically, while the program is executed on some concrete inputs.
- Concolic: Concrete + symbolic.
- Concolic testing maintains a concrete state and a symbolic state.
- Concrete state is the normal notion of state of a program: values of all variables in the program.

So, one of the techniques that people have introduced that helps us to overcome the disadvantages of symbolic testing is what is called concolic testing. Concolic testing is short form for concrete plus symbolic: con colic. That is how they have coined the term, it means that you instead of keeping a symbolic value alone which we do in symbolic testing you keep the concrete value along with the symbolic value. We will learn one particular concolic testing technique called DART directed automated random testing. I will teach that to you in the next few lectures. We will see how it overcomes several of the disadvantages that we have listed of symbolic testing in this slide.



Concolic testing as I told you, keeps the concrete state along with the symbolic state. Concrete state is the normal notion of a state of a program, if you remember in the last lecture.

(Refer Slide Time: 14:20)

Example 1

Consider the following program fragment:

```
1 Sum(a,b,c)
2   x = a+b ;
3   y = b+c;
4   z = x+y-b;
5   return(z);
6 end
```



If you would let me go back in my slides for a minute, we had this sum program if you remember, this program that calculated the sum.

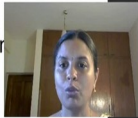

(Refer Slide Time: 14:22)

Normal execution of Sum on inputs

Normal execution of program Sum on inputs 1, 3, 5 is given in the table below:

After stmt.	x	y	z	a	b	c
1	?	?	?	1	3	5
2	4	-	-	-	-	-
3	-	8	-	-	-	-
4	-	-	9	-	-	-
5	returns 9					

In the above table, - represents unchanged values and ? represents undefined (uninitialized) values.





A concrete state would be value of a is 1 b is 3, c is 5, x is 4, y is 8, z is 9. After statement number 4 concrete state would be x 4, y 8, z 9, a 1, b 3, c 5.

(Refer Slide Time: 14:40)

Symbolic execution of Sum

After stmt.	x	y	z	a	b	c	PC
1	?	?	?	α_1	α_2	α_3	true
2	$\alpha_1 + \alpha_2$	-	-	-	-	-	-
3	-	$\alpha_2 + \alpha_3$	-	-	-	-	-
4	-	-	$\alpha_1 + \alpha_2 + \alpha_3$	-	-	-	-
5	Returns $\alpha_1 + \alpha_2 + \alpha_3$						

The above table represents symbolic execution of $\text{Sum}(\alpha_1, \alpha_2, \alpha_3)$. Path Condition/Constraint is abbreviated as PC.



After symbolic execution in line number 4, x would be $\alpha_1 + \alpha_2$ symbolic expression, y would be $\alpha_2 + \alpha_3$ another symbolic expression, there would be another symbolic expression for z, the inputs a, b and c would have gotten symbolic values. So, now, I will move forward to concrete testing.

So, concrete testing or concolic testing keeps concrete state as I just told you which means it keeps concrete values; actual values and it will keep a symbolic state, it will keep the symbolic values and the expressions. Why does it keep both? The difference will be very obvious very soon. I will tell you and it what it tries to do is that by keeping track of both it has the convenience of moving back and forth. So, that is what it does.

(Refer Slide Time: 15:31)

Concolic execution: Example 2

- Concolic execution generates a random input: $\{x = 22, y = 7\}$ and executes the program both concretely and symbolically.
- Concrete execution takes the “else” branch; symbolic execution generates the PC $x_0 \neq 2y_0$.
- Concolic testing negates PC, solves $x_0 = 2y_0$ to get test input $\{x = 2, y = 1\}$, forcing the program along a different execution path.
- Next concolic execution generates PC $(x_0 = 2y_0) \wedge (x_0 \leq y_0 + 1)$.
- This PC is negated, constraint $(x_0 = 2y_0) \wedge (x_0 > y_0 + 10)$ is solved to get test input $\{x = 30, y = 15\}$, error state is reached.
- After three executions, all program paths have been explored.



So, if I have to take the other example that I told you in the last class, again let me go back to show you that example.

(Refer Slide Time: 15:40)

Example 2, re-cap

Consider the program in example 2 again:

```
1 int twice(int v) {
2     return 2 * v;
3 }
4 void testme(int x, int y) {
5     z = twice(y);
6     if(z == x) {
7         if(x > y + 10)
8             ERROR;
9     }
10 }
11 int main() {
12     x = sym_input();
13     y = sym_input();
14     testme(x, y);
15 }
```



This was that example, if you remember, it did took 2 inputs x and y, did z as twice(y) and then it said z = x, if it is, then $x + y > 10$ then there is an error, this was the program. Here if you remember the symbolic expression was all these, right.

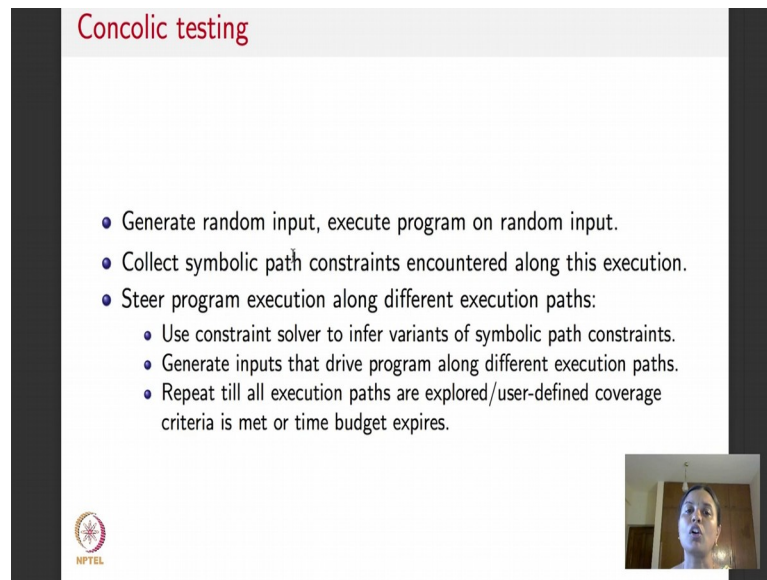
I will show you that slide also, symbolic state was x is x_0 , y is y_0 , z is $2 y_0$. These were the path constraints $x_0 = y_0$, $x_0 \neq 2 y_0$ and this was after the second if statement was executed.

What we will now do is look at concolic statement. So, concolic execution, what will it do? It will keep a concrete state. How will it keep a concrete state, here is one particular way of keeping a concrete state. It will give a random input to the program. Let us say $x = 22$, $y = 7$ or any other choice. What it will do is that it will start executing the program. Once it starts executing the program, $x = 22$, $y = 7$. Go back to the code of the program. For this concrete input, this condition will it pass? It will not pass. So, this condition will go as fail which means the second if statement will not be evaluated.

So, that is what is written here. The concrete execution on this input will make the program take the else branch. Symbolic execution will generate saying the by the way, the program took an else branch by because $x_0 \neq y_0$. What will concrete, concolic testing now do is that it says the program took the negation of this, right, I will negate this condition to see what the program will do if it takes the other one. So, it will negate this which means it will say x_0 , give inputs x and y such that $x_0 = 2 y_0$. Here is such input x is 2, y is 1. Now what will it do? This it will go to the next statement in the program.

The next statement in the program concolic testing will say I did this which was already my path condition. By the way I encountered one more if statement which was this and this is the part of the if statement that is satisfied by this test case. Now it will say it took one particular branch. I will take this condition, negate it to make it, there take it the other branch. So, when it negates it, it will get this condition $x_0 = 2 y_0$; that is not the predicate under focus, it will keep it as it is. This was the predicate under focus, here it was $x_0 \leq 2 y_0$. I will say $x_0 > y_0 + 10$. Sorry, there is a typographical error here it should be 10 please read it as 10. So, now, it will correctly get the test input $x = 13$, $y = 5$ and it will hit the error statement in the program. So, this is how concolic execution works? This is just to meant to be an introductory example, I will explain concolic execution. In fact, this particular technique of concolic execution called DART, in detail to you.

(Refer Slide Time: 18:46)



The slide is titled "Concolic testing" in red text at the top. It contains a bulleted list of steps for the process. In the bottom right corner, there is a small video inset showing a person speaking. The NPTEL logo is in the bottom left corner.

- Generate random input, execute program on random input.
- Collect symbolic path constraints encountered along this execution.
- Steer program execution along different execution paths:
 - Use constraint solver to infer variants of symbolic path constraints.
 - Generate inputs that drive program along different execution paths.
 - Repeat till all execution paths are explored/user-defined coverage criteria is met or time budget expires.

So, what is concolic testing do? This is you can think of this as a concise summary of concolic testing. It has a program that it wants to test. What it will first do it will generate a random test input to the program. It will execute the program on the random test input. The program will take certain program paths. As the program takes certain program paths it will parallely symbolically execute the program. When it symbolically executes the program, it will collect constraints which said the program encountered the first if statement. The first if statement encountered in the program for this random input turned out to be true.



So, it will keep that constraint over symbolic variables and then it will move on. Let us say program encountered a second if statement second if statement on this run of the program turned out to be false. So, it will take the constraint corresponding to the first if statement negate the constraint corresponding to the second if statement, AND these two. It will keep collecting path constraints like this. Now this represents one concrete execution of a program on that random input and the whole set of path constraints. Now what concolic testing will do is that this path constraint is an AND of several different path constraints, right. So, it will systematically negate each clause in it and then try to see what the new execution of the program does, and for that it will use a constraint solver to get concrete values of inputs and it will repeat this till all the execution paths are done.

(Refer Slide Time: 20:19)

Example 4

Consider the function `h` in the program segment below:

```
int f(int x)
{ return 2 * x; }
int h(int x, int y) {
    if (x != y)
        if (f(x) == x + 10)
            abort(x);
    return 0;
}
```

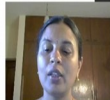



So, here is another example of concolic testing. So, this is a different function. So, it says there is a function `f` which says `int x` is an integer variable. Like the earlier thing it just did `2 * x`, there is a function `h` which says it takes 2 arguments `x` and `y` and returns an integer. It has 2 if statements first one if $x \neq y$, then it goes to the second statement second says if `f(x)` which is this `f` up here equal to `x + 10`, then you abort. Abort means you stop, there is an error in the program you return 0. If you notice that it is very similar to the earlier one; just a small piece of difference from the earlier example that we had.

(Refer Slide Time: 21:08)

Concolic testing on Example 4

- Reaching the defective function `h` is difficult using a randomly generated test input.
- Using concolic, we can reach the error state in two execution steps.
- Randomly generate inputs: $x = 26, y = 34$.
- Predicates $x_0 \neq y_0$ and $2x_0 \neq x_0 + 10$ are formed during the concolic execution.
- Concolic testing calculates the PC $x_0 \neq y_0 \wedge x_0 = x_0 + 10$ by negating the last predicate of the earlier PC, this leads to the error statement.


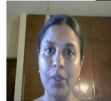


Now, the problem is this is the place that I want to reach. Using concolic testing, how will I do? I will first randomly generate 2 inputs, let us say x is 30, y is 34 any randomly generated inputs. Then 2 predicates are there, $x_0 \neq y_0$ which I know because in this random input that is true and then $x_0 \neq x_0 + 10$. That is what this means right, because $f(x)$ does $2 \times x$ and $x + 10$ is captured here. So, both these are taken. Path constraint looks like this, I take the second one, negate it then I will get something that will reach the statement. So, this will be true and this will be true, I would have reached the error. That is how concrete testing works, is that clear?

(Refer Slide Time: 21:54)

Symbolic testing techniques: State of the art

- Modern constraint solvers can solve complicated non-linear functions, improving symbolic testing techniques.
- Many years of research spanning over several papers have extended these techniques to handle pointers, files, multi-threaded programs, dynamic data structures etc.
- Some popular tools are CUTE (for C, UIUC), jCUTE (for Java, at UIUC), PEX (for .NET, available as a Visual Studio add-in), SAGE (Microsoft).
- Several other companies have developed tools using such techniques, NASA, IBM, Fujitsu etc.

I will explain it using an example and look at the general techniques of dart and so on in the next lecture. Before I stop, I would like to tell you that there are modern constraint solvers. I will give you some references when I begin the next lectures. Now there are several popular symbolic testing techniques tools that exist, there is a tool called CUTE symbolic testing you in a concolic unit testing engine for C developed by University of Illinois. This CUTE for Java which is again developed by University of Illinois at Urbana Champaign; the PEX which is available for dotnet code as a part of Microsoft visual studio the sage and so on and several other companies have developed symbolic testing techniques tools I will give you the URLs of all these and some constraints all words when I begin the next lecture. In the next lecture, what we are going to see is this dart technique in detail directed automated random testing, I will tell you what that is and how concolic testing works on that.

Thank you.

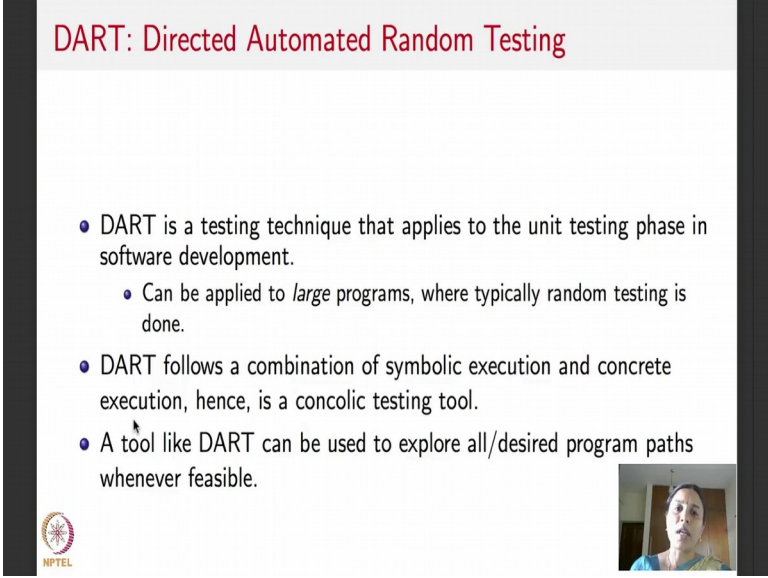
Software testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 52
DART: Directed Automated Random Testing

Hello again, we are going to continue with viewing symbolic testing techniques. Last class I told you about symbolic testing, then we moved on and looked at the disadvantages of symbolic testing. One of the biggest disadvantages is the solving the path constraint. If I collect a set of path constraints, I may not be able to solve it because satisfiability problem is in general un-decidable. The current day solvers that solve these path constraints can handle linear path constraints very well. But let us say if you had something like $x^2 > 0$ or $\log^2 x > 0$ which we could encounter while writing code; then the solutions to these become difficult. As a solution, we presented a hybrid symbolic testing technique called concolic testing: concolic, a short form for concrete plus symbolic and as I told you, it does both concrete and symbolic execution together.



So, today what I am going to tell you is one particular concolic testing tool called DART stands for directed automated random testing. We will look at this algorithm that DART deploy is to do test case generation using symbolic testing in detail. We will do it over 3 lectures. In this lecture, I will introduce you to DART. We will get started on what exactly DART does and will look at a mix of concrete and symbolic execution in DART. In the next lecture, I will walk you through the precise algorithm that DART deploy is to instrument a program and to write the test driver. In the last lecture on the DART which will be the third lecture from now, we will look at an example C program where a tool like DART can be applied to do concolic testing.

(Refer Slide Time: 01:59)



DART: Directed Automated Random Testing

- DART is a testing technique that applies to the unit testing phase in software development.
 - Can be applied to *large* programs, where typically random testing is done.
- DART follows a combination of symbolic execution and concrete execution, hence, is a concolic testing tool.
- A tool like DART can be used to explore all/desired program paths whenever feasible.

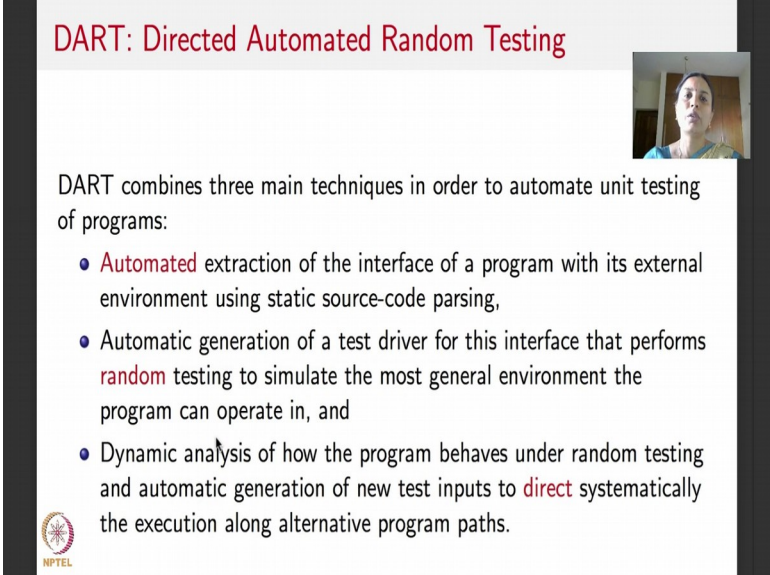
 

So, what is DART? DART is basically a testing technique or an algorithm that applies to unit testing phase. Let us say you have a fairly large piece of code and you are the developer or the programmer and you have been asked to unit test the code. We have seen several testing techniques to do unit test the code, but let us say you are unable to generate graph models or logic models and you have a fairly large piece of code, let us say a few 100-1000 lines of code with you to test. Typically as a unit tester, the burden will be on you to write the test driver to instrument the program for which they may or may not be time left. So, typically unit testers resort to what is called random testing which is they generate a randomly generated test vector a few of them, run the program on it if they are lucky, they might find an error, but if they are not lucky then they may not find an error.

So, what we will see today is a tool called DART which will let you do unit testing in a reasonably automatic way, do not have to write drivers, do not have to instrument the program yourself, DART will do it all on its own. We will see DART as it applies for C, but these days DART like tools are available for Java, for dot net and for several other frame works as I told you. So, what is DART do? DART follows concolic testing which means, it does both symbolic execution and concrete execution and the advantage of DART is that it can be used to explore all feasible program paths, which means it can be done to do what is called exploratory, all combinations paths testing. We have seen when we did graph based testing that all complete path coverage is infeasible several times. So,

DART also may not be able to achieve complete path coverage, but it will do so whenever it can and we will see when it can and when it cannot.


(Refer Slide Time: 03:54)



DART: Directed Automated Random Testing

DART combines three main techniques in order to automate unit testing of programs:

- **Automated** extraction of the interface of a program with its external environment using static source-code parsing,
- Automatic generation of a test driver for this interface that performs **random** testing to simulate the most general environment the program can operate in, and
- Dynamic analysis of how the program behaves under random testing and automatic generation of new test inputs to **direct** systematically the execution along alternative program paths.



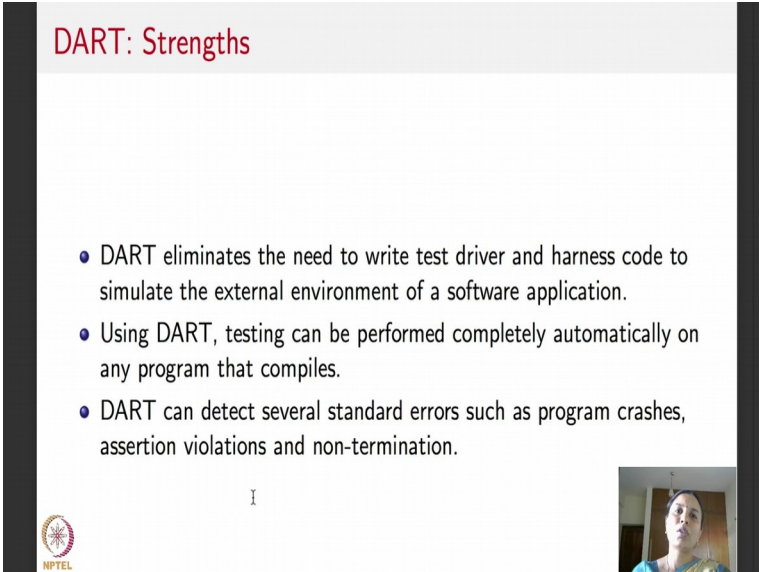
So, what does DART do? DART uses basically 3 main techniques to do its directed automated random testing. So, what are the 3 techniques that DART uses? DART, as I told you, you just give it an executable program, DART will write the driver for your instrument, the program run it and tell you whether it is found an error, explored all paths or it is unable to do either of these. It will do all these automatically. So, to or do all this automatically, DART uses 3 techniques to automate unit testing. First that it does is that it extracts the interface of a program with its external environment automatically. What do we mean by extracting an interface of a program with its external environment? A program might take input from the user, program might send outputs to the console, program might read data from a database.

So, all these things which the program talks to or interfaces which are called external interfaces. DART first does static source code parsing and automatically extracts these interfaces of a program and then what does DART does is it does, writes the test driver. What is a test driver? If you remember from our lessons and integration testing, it is a program that takes and executes your main program under test, DART generates the test driver automatically for this interface which is basically the interface that executes the program, supplies it inputs and what is the main job of the test driver generated by

DART. It is a randomly test a program; randomly test a program means to give random inputs to the program in test. We will see what it means very soon and finally, what is DART do? DART dynamically analyzes how the program behaves under this random testing that was done with to start with in step 2. And then by systematically manipulating the behavior of the program under the random test, DART will generate new inputs to direct the program for specific program paths.

So, let me repeat the 3 steps of DART. The first step is, DART figures out automatically what are the interfaces the program has. After it does that, DART writes a test driver, again automatically that picks up random test case from the interface and starts executing the program on the random test case. As the program executes the random test case, DART collects a few pieces of information. Using that pieces of information, DART will now generate newer and newer inputs that will systematically explore the program on paths, around the paths that the program took on the random test case.

(Refer Slide Time: 06:43)




DART: Strengths

- DART eliminates the need to write test driver and harness code to simulate the external environment of a software application.
- Using DART, testing can be performed completely automatically on any program that compiles.
- DART can detect several standard errors such as program crashes, assertion violations and non-termination.

NPTEL

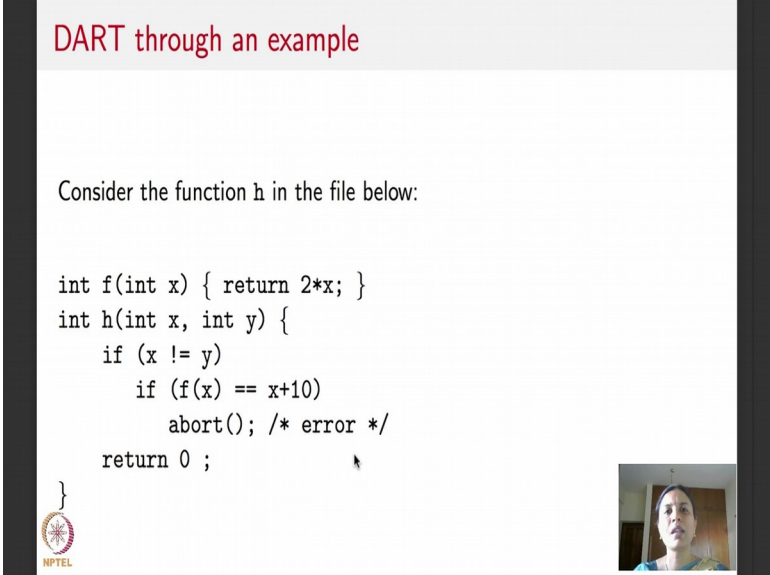
I



So, before we move on and understand DART, what are the strengths of DART, because DART does this interface extraction and driver writing automatically, the user or the tester does not have to write the test driver and the harness code to stimulate the external environment of a software application and because of that testing can be done completely automatically on any program that complies. We will do it for C through an example. DART can detect several standard errors like program crashes that happen because of

errors like division by 0, violations of assertions, presence of non terminating loops and so on.

(Refer Slide Time: 07:21)



DART through an example

Consider the function h in the file below:

```
int f(int x) { return 2*x; }
int h(int x, int y) {
    if (x != y)
        if (f(x) == x+10)
            abort(); /* error */
    return 0 ;
}
```

NPTEL

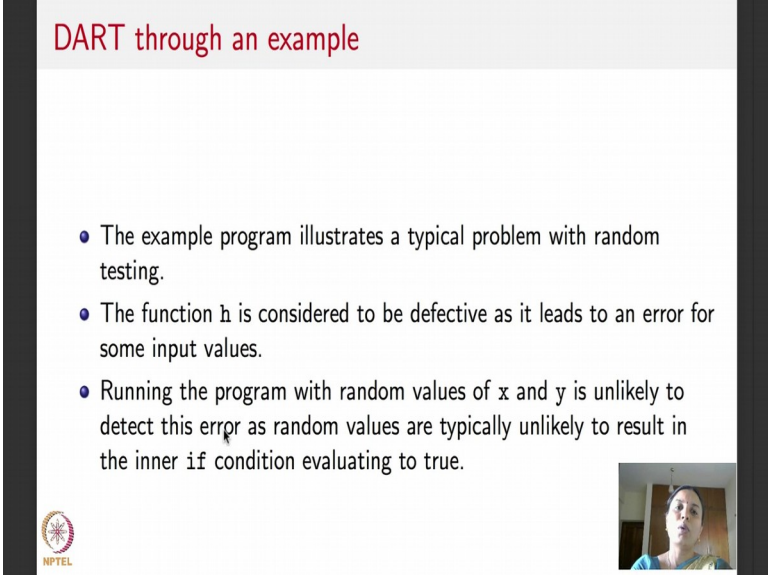
So, before we understand DART, let me revisit the example that I had presented to you towards the end of last lecture. Here is a function h, h takes 2 arguments; integer arguments x and y and returns an integer. It has 2 if statements inside it, first one says if x is not equal to y, you go to the second if statement. Second if statement calls a function f, function f code is given here above function f basically takes an argument x and returns 2 x. So, it says if f of x is equal to x plus 10 which means if 2 x is equal to x plus 10, then read it as that this program for h has reached an error state. So, there is a word called abort and it returns 0.

So, now let us say, this is written by a particular developer and his or her job is to unit test this code. So, typically developer will resort to random testing. Let us assume that because the code is fairly large. In this case obviously, it is not a large example, but let us say that assume that the developer has a large code, he or she will resort to random testing. Random testing means what generate random values of inputs for x and y. Let us say you generate some value, x is 25, y is 35, something like that. Now where is the error in the program? The error in the program is here nested inside the 2 if statements. The conditions for both the if-s should pass for me to be able to reach error. Using a randomly generated input with high probability I will definitely pass the first if

condition, it is unlikely that the 2 randomly generated inputs x and y will turn out to be the same.

So, they will most likely be different. So, I will pass the first if condition, but the second if condition is very specific. It says f of x which is $2x$ is x plus 10, then the second if condition passes. It is not very likely that the randomly generated input will pass the second condition. So, no matter how many times you randomly generate an input and test a program, you might not hit the error statement which is present here in the program. We will see how DART will through one random generation of test case will systematically hit the error statement in the program.

(Refer Slide Time: 09:36)



DART through an example

- The example program illustrates a typical problem with random testing.
- The function h is considered to be defective as it leads to an error for some input values.
- Running the program with random values of x and y is unlikely to detect this error as random values are typically unlikely to result in the inner if condition evaluating to true.


NPTEL

Video inset showing a person speaking.


So, as I told you, this example program illustrates a typical problem with random testing. The function h is considered to be defective because it leads to an error for some input values of x and y . In particular, x should not be equal to y and f of x should be equal to x plus 10; that is when the input it is an error. Running a program with random values for x and y is highly unlikely to find the error because the inner if condition is highly unlikely to evaluate true for random values of x and y .

(Refer Slide Time: 10:05)

DART on the example



- DART guesses the value 354 for x and 34567 for y .
- As a result, it executes the then-branch of the first if statement, but, fails to execute the then-branch of the second if statement and hence no error is encountered.
- Intertwined with the normal execution, the predicates $x_0 \neq y_0$ and $2 \cdot x_0 = x_0 + 10$ are formed on-the-fly.
- x_0 and y_0 are symbolic variables that represent the values of the memory locations of the variables x and y .
- Note the expression $2 \cdot x_0$; it is formed through an inter-procedural dynamic tracing of symbolic expressions.



So, how will DART now find the error? What are the 3 steps of DART? Figure out the interface of the program begin, generate a random input to the program run the program on it, collect some details, then systematically manipulate the details that you have collected to direct the program on specific program paths. So, DART will begin by generating a random value. Let us say it guesses 354 for x and some 34567 for y , some values. Now what will happen? Then in this code, it will obviously, pass the first if statement because x is not equal to y , it will go. It will fail the second if statement because $2x$ is not equal to $y + 20$. So, DART will say that the first if statement is passed, the then branch of the second if statement was not taken so, the error is not encountered.


This is a normal execution intertwined with normal execution DART also does extra book keeping, one of the book keeping it does is to collect the predicates that evaluated to true and the predicates that evaluated to false. For this randomly generated test input, the first predicate x is not equal to y evaluated to true. So, if DART remembers this, it does not remember it as x is not equal to y , it remembers it as x_{naught} is not equal to y_{naught} . What is this x_{naught} and what is this y_{naught} ? They are symbolic inputs representing x and y . And then the second path constraint is $2x_{\text{naught}}$ which is false which is actually going to the function f , says compute $2x$, so, you substitute it back here x is symbolically represented by x_{naught} . So, you get $2x_{\text{naught}}$ is the same as x_{naught}

plus 10. As I told you x naught and y naught are symbolic variables that represent the value of the memory locations corresponding to the values of the variables x and y .


So, this will give you a clue of what symbolic representation is. To represent x symbolically, give a name to the memory location of x . To represent y symbolically give a name to the memory location of y . In this case we have given symbolic value for x is x naught, symbolic value for y is y naught. Now another thing to note is that how did this path expression come. If you go back to the code this path expression came from if condition. This if condition actually had f of x ; that means, that DART had to go to the code corresponding to f understand that f of x is $2 \star x$, substitute it back here and then say this is the constraint. So, this comment here just says that. It says that the path expression $2 \star x$ naught in this predicate is formed through as inter-procedural dynamic tracing of symbolic expression, because from h , the control gets transferred to the function f from where you understand that it is $2 \star x$ and then you substitute it back. DART can do this automatically is what is being said.

(Refer Slide Time: 13:02)

DART: Exploring different program paths



- DART dynamically gathers knowledge about the execution program, they call it **directed search**.
- Starting with the execution of the program on a random input, DART calculates during each execution, an input vector for the next execution.
- This vector contains values that are the solution of symbolic constraints gathered from predicates in branch statements during the previous execution (called **path constraints**).
- The new vector attempts to force the execution of the program through a new path.
- By repeating this process, a directed search attempts to force the program to sweep through all its feasible execution paths.



Now, DART has done one random generation of a test vector for x and y , collected these symbolic constraints. Now DART wants to explore, so, this means what the program for this randomly generated input for x and y has taken one particular execution path. Now DART wants to systematically explore the neighborhood of this execution path. So, what it will do is it will do what is called a directed search. So, what it starts with the


execution of the program on the random input and calculates an input vector for next execution. How does it do? As I told you when the program runs on the random input DART collects these path constraints, right. So, what it will do is that it will keep these paths constraints, let us say in a stack. The first constraint is this the second constraint is this; this constraint passed this constraint failed, it will keep that information.


Now, DART wants to explore the neighborhood, for this randomly generated input the second condition failed. So, now, what it will try to do? It will try to negate the second condition that failed, which means it will try to make the second condition pass. So, it will generate a path constraint which will negate the second, this thing, and it will give it to a constraint solver. The constraint solver will give input values for x and y that will satisfy the first constraint and that will satisfy the second constraint.

(Refer Slide Time: 14:30)

DART on the example

- For the example program, the path constraint $\langle x_0 \neq y_0, 2 \cdot x_0 \neq x_0 + 10 \rangle$ represents an equivalence class of input vectors containing all input values that drive the program through the path that was just executed.
- To force the program through a different equivalence class, DART-instrumented h calculates a solution to the path constraint $\langle x_0 \neq y_0, 2 \cdot x_0 = x_0 + 10 \rangle$ obtained by negating the relevant predicate on the current path constraint.
- A solution to this path constraint (for example, $(x_0 = 10, y_0 = 4566)$) is recorded and when the instrumented h runs again, it reads the recorded values. Execution with these values reveals the error in the program.

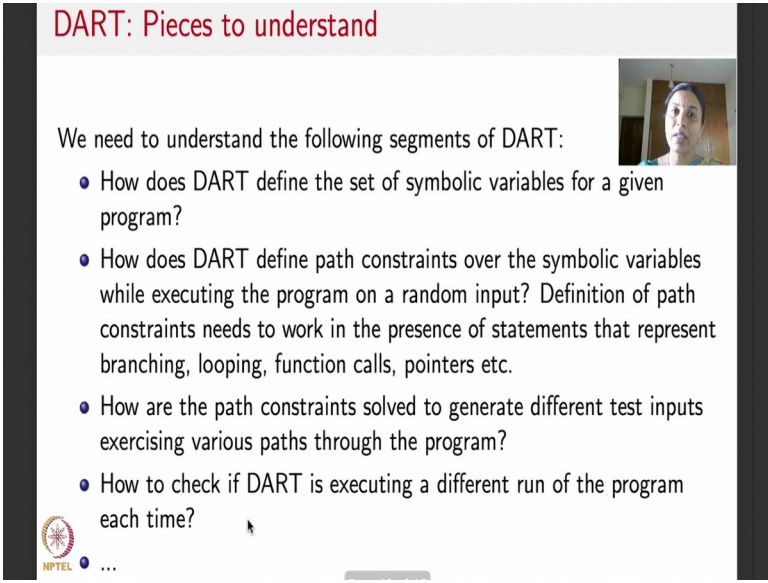




So, that is what is written here. For the example path on the random input these were the constraints that were satisfied: x naught is not equal to y naught, I mean not satisfied sorry, generated. x naught is not equal to y naught and $2x$ naught is not equal to x naught plus 10. Now what DART will do it will take the second path predicate and negate it which means they did it will get $2x$ naught is equal to x naught plus 10. What it will do is it will try to give this as a path constraint to a constraint solver and ask the constraint solver to give it values for x naught and y naught that will satisfy this. Constraint solver could give a value something like this.

Let us say it will say x naught is 10, y naught is something, and now it will check. X naught is not equal to y naught, correct, 2 x naught is x naught plus 10 both conditions pass. So, which means what both if-s pass the program is reached the error statement. This is how DART goes about systematically exploring neighboring program paths and in the process if it reaches an error statement, it stops and says I have found the error in the program.

(Refer Slide Time: 15:31)



DART: Pieces to understand

We need to understand the following segments of DART:

- How does DART define the set of symbolic variables for a given program?
- How does DART define path constraints over the symbolic variables while executing the program on a random input? Definition of path constraints needs to work in the presence of statements that represent branching, looping, function calls, pointers etc.
- How are the path constraints solved to generate different test inputs exercising various paths through the program?
- How to check if DART is executing a different run of the program each time?

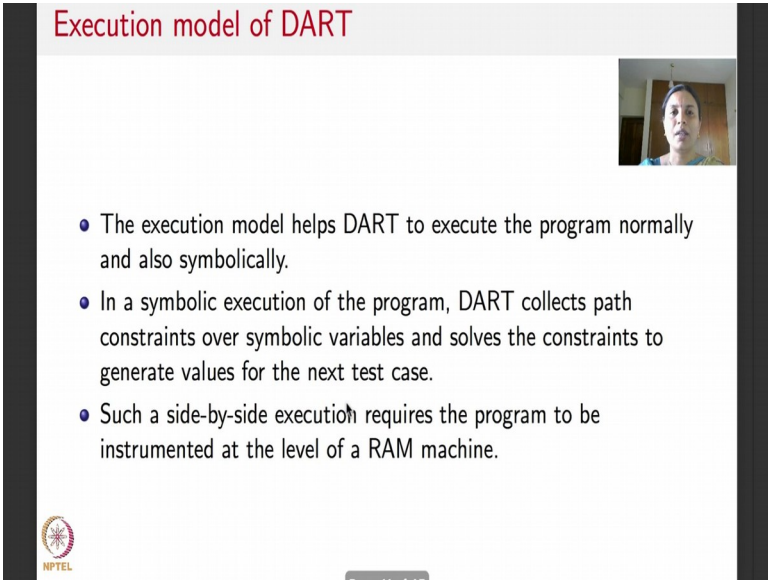
NPTTEL ...

So, that was DART through an example. As I told you our goal is to understand DART in detail. So, what are the pieces that we are going to understand? These are the questions that we will try to answer in this lecture and partly in the next lecture. We will understand how DART defines symbolic variables for a given program. For this example, we will understand how DART came up with x naught, y naught; how did it do it? The second is how does DART define path constraints over the symbolic inputs while it executes the program on a random input, which means what? Here, it executed a program on a random input and it generated these 2 path constraints corresponding to the path taken by the program on the random input. How does DART do that? This toy example did not have much features.

But I my claim is DART will work on arbitrary C programs, which means what they could have branching, looping, they could have function calls which we already saw in this example they could have pointers and so on. So, DART is should be able to generate

these path constraints in the presence of all these entities. The next part is how does it flip one of the conditions in the path constraints? Flipping means, how does it negate one of the conditions in the path constraints. How does it give it to a constraint solver get a solution and then run the program again on that solution and how do you know that DART is actually executing a different run. So, these are the kind of questions that we will understand in detail.

(Refer Slide Time: 17:11)



The slide is titled "Execution model of DART" in red text. It features a small video inset in the top right corner showing a person speaking. The main content consists of three bullet points:

- The execution model helps DART to execute the program normally and also symbolically.
- In a symbolic execution of the program, DART collects path constraints over symbolic variables and solves the constraints to generate values for the next test case.
- Such a side-by-side execution requires the program to be instrumented at the level of a RAM machine.


In the bottom left corner, there is a small circular logo with the text "NPTEL" below it.

So, before, for us to be able to understand in detail, we need to understand what the execution model of DART is. Which means how does DART go about doing interface extraction, instrumentation, writing a driver. How does DART do this? Remember one thing, DART executes a program both concretely or normally and symbolically. While executing a program symbolically DART collects the symbolic path constraints and solves these constraints by using a constraint solver as a black box. For normal execution, it just records the values of the variables that were defined as the program executes. Now it has to execute both symbolically and normally, this side by side execution of a program demands that DART work, instrument the program at the level of a RAM machine, which means what we are going to assume that the program is actually residing on a particular architecture of a RAM and look at how the program will look like as its residing in memory.

(Refer Slide Time: 18:16)

Execution model of DART: Symbolic variables

- **Memory** \mathcal{M} : a mapping from memory addresses m to say 32-bit words.
 - $+$ denotes updating of memory. For e.g., $\mathcal{M}' := \mathcal{M} + [m \mapsto v]$ is the same as \mathcal{M} except that $\mathcal{M}' = v$.
- **Symbolic variables** are identified by their addresses.
- In an expression, m denotes either a memory address or the symbolic variable identified by address m , depending on the context.
- A **symbolic expression** or just an expression, e is given by
 - m , c (a constant), $*(e, e')$ (multiplication), $\leq (e, e')$ (comparison), $\neg e'$ (negation), $*e'$ (pointer dereference) etc.
- Symbolic variables of an expression e are the set of addresses m that occur in it.
- Expressions have no side effects.




We will understand various bits and pieces of DART. If you focus on the title slide, first thing I am going to understand is symbolic variables. What are symbolic variables in the execution model of DART? We will understand the semantics of a program after that in the execution model of a star DART, followed by understanding of statement labels, concrete semantics, inputs and program execution.

(Refer Slide Time: 18:30)

Execution model of DART: Semantics of a program

- Typical semantics of a program:
 - A transition system where a state represents the values of all variables and a program counter and transitions represent execution of a program statement resulting in change of state.
 - Each execution of the program results in a path through this transition system.
- For DART, we need to define semantics of a program at memory level. We define it similar to the typical semantics. At memory level, **statements** are specifically tailored abstractions of the machine instructions that are actually executed.



(Refer Slide Time: 18:35)

Execution model of DART: Statement labels

- For denoting statement labels, we use a set of labels that denote instruction addresses.
- If l is the address of a statement (other than **abort** or **halt**) then $l + 1$ is also an address of a statement. There is an initial address, say, l_0 .
- Statements could be of various kinds:
 - A **conditional statement** c of the form **if** e **then goto** l' , where e is an expression over symbolic variables l' is a statement label,
 - an **assignment statement** a of the form $m \leftarrow e$, where m is a memory address,
 - **abort** corresponding to program error, and **halt** corresponding to normal program termination.
- A function *statement* – $at(l, \mathcal{M})$ specifies the next statement to be executed.

So, these are the bits and pieces with which we will understand the execution model of DART.

(Refer Slide Time: 18:43)

Execution model of DART: Program Execution

- Let \mathbf{C} be the set of conditional statements and \mathbf{A} be the set of assignment statements in P .
- A **program execution** w is a finite sequence in $\mathbf{Execs} := (\mathbf{A} \cup \mathbf{C})^*(\mathbf{abort}|\mathbf{halt})$.
- They further simplify program execution and view w as being of the form $\alpha_1 c_1 \alpha_2 c_2 \dots c_k \alpha_{k+1} s$ where $\alpha_i \in \mathbf{A}^*$, $c_i \in \mathbf{C}$, and $s \in \{\mathbf{abort}, \mathbf{halt}\}$.
- An alternate view is to consider $\mathbf{Execs}(P)$ is a tree, with assignment nodes having one successor, condition nodes have one or two successors and leaves being labeled by **abort** or **halt**.
- Each input vector results in an execution sequence, as a path in this tree.

We begin with symbolic variables. As I told you, what are symbolic variables? Symbolic variables are place holders for actual values of a variable. In a program as it runs in a real environment which is the best entity to represent a place holder. The best entity to represent a place holder is the actual memory location. Let us say there are 2 variables as we saw in the example, x and y . The symbolic representation of x is the memory location

for x , the symbolic representation of y is the memory location for y . So, first we need to understand how memory looks like. Let us say you have a 32 bit processor or a 64 bit processor or a 128 bit processor then, what is memory? Memory for a 32 bit processor can be thought of as a mapping from memory addresses, call it M , to 32 bit words. What is $+$ denote? It denotes that the memory is updated. For example, suppose I write something like this it means that M' is the same as memory M except that the memory location small m is updated with the value for V .

Otherwise all other memory locations in m prime M' are the same as that of M . What are symbolic variables? As I told you symbolic variables are defined by their addresses. Now in an expression, M will denote either a memory address or a symbolic variable identified by that address depending on the context as we move on, this will become clear. Now what is a symbolic expression? Symbolic expression is the same as our arithmetic and logical expressions as we saw in the last lecture, but instead of working over normal variables, it works over symbolic variables. So, symbolic expressions could just be variable names which have memory locations for us, it could be a constant which is also another memory location, it could be an arithmetic expression. Please read this star e , e prime within brackets multiplication as any arithmetic expression.

So, it could be plus; it could be minus, it could be slash, it could be mod. For the sake of simplicity, only star is multiplied. But this is to be read and understood as a generic instantiation of any arithmetic expression. Similarly this less than or equal to (e, e') should be read and understood as a generic instantiation of any relational operator which compares 2 expressions, 2 variables e and e' . It could be less than or equal to, greater than or equal to, not equal to, less than, greater than and so on. Again this not e prime negation of e prime, is to be interpreted as representing one instances of a logical operator can be substituted with any logical operator and because we are using it for C , we also work with pointers the star e prime.

So, this star is a binary star which is multiplication. This star is a unary star which reference to which refers to pointer dereferencing. So, basically symbolic expression is a lot like our arithmetic, logical, relational expression. Instead of working over normal variables, it works over symbolic variables. So, it has constants, it has symbolic variables which are represented by memory location, it has addition, subtraction, multiplication,

division, mod, it has all the relational operators, all the logical operators and operators dealing with pointer arithmetic. We assume that expressions have no side effects.

Now, the semantics of a program. Typically when we look at program, we understand what is the semantics of a program? What is the state of a program? State of a program is the values of all variables in the program plus a location counter which tells you which is the statement that the program is executing. So, that is usually given in this transition system, where a state represents, as I told you a tuple of all the values of a program and variable in a program counter and transition means what. One statement executes and the transition tells you; how one state changes to another state; which value of a variable changes to which other value.

So, each execution of a program results in a path to this through this transition system. But please remember because we are working with DART and DART needs to both concretely and symbolically execute a program, we need to define the semantics of a program at the level of a memory. Which means what, we define it similar to statement, the semantics that is given up here, but we say statements are nothing, but simple machine instructions because that is what happens at the level of a memory.

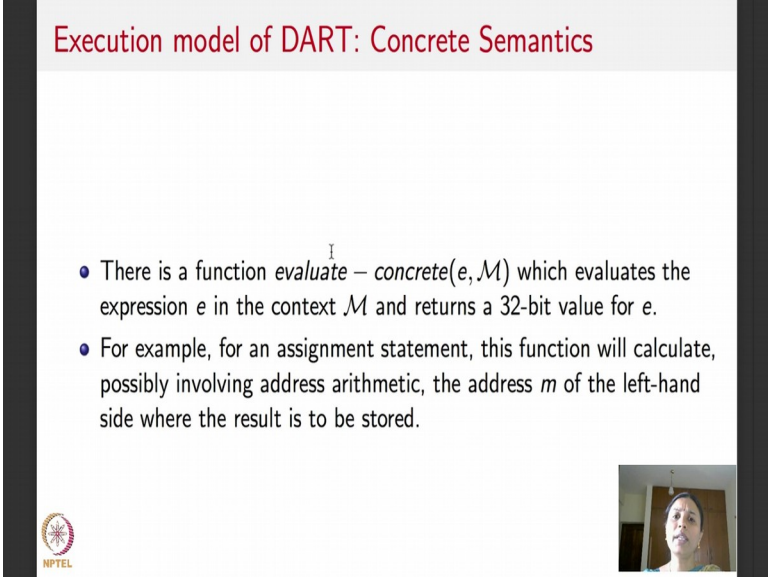
So, how are statement labels denoted? We take directly the instruction addresses that are available in our computer architecture and say those are the set of statement labels. So, if n is the address of a statement, let us say a statement label then $n+1$ is also an address of a statement. As I told you DART is meant to work on our all C programs, but for the purpose of illustrating the algorithm of DART, what the authors of the paper do is consider restricted syntax.

So, what they say is that for now simplicity assume that program has only the following condition kinds of statements, it has conditional statement called C which is like a typical if statement. If an expression evaluates to true then go to a particular statement l primewhere e is a symbolic expression and l is a label of a statement. We also assume that in our restricted syntax a program has only assignment statements which assigns an expression to a memory address and in addition to this, the program has a special statement called abort, which we saw in our example and another statement called halt which corresponds to normal program execution. All other statements that you will

typically find in programming languages like while, anything else, can be expressed using these.

So, for simplicity we assume that as far as understanding the execution of DART is concerned, DART has only, the programming language like C on which DART works, has only condition statements assignment statements and special statements like abort and halt. We also assume that there is a special function that is available to us, let us call it statement at $l\ m$, which specifies what is the next statement to be executed.

(Refer Slide Time: 25:12)



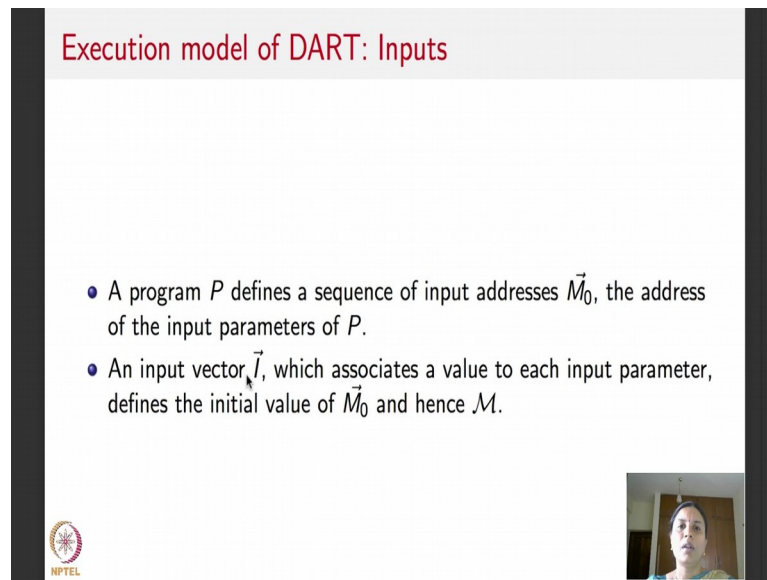
The slide is titled "Execution model of DART: Concrete Semantics" in red text. It contains two bullet points:

- There is a function $evaluate - concrete(e, \mathcal{M})$ which evaluates the expression e in the context \mathcal{M} and returns a 32-bit value for e .
- For example, for an assignment statement, this function will calculate, possibly involving address arithmetic, the address m of the left-hand side where the result is to be stored.

In the bottom left corner, there is a small circular logo with the text "NPTEL" below it. In the bottom right corner, there is a small video inset showing a person speaking.

Now, concrete semantics of DART, what will it do? It will directly give you the actual value that is stored in the memory location corresponding to every variable. For example, for an assignment function, this function will calculate, this evaluate concrete which evaluates the concrete semantics, will calculate using some address arithmetic, the address of the left hand side where the result is to be stored.

(Refer Slide Time: 25:33)



The slide is titled "Execution model of DART: Inputs" in red text. It contains two bullet points:

- A program P defines a sequence of input addresses \vec{M}_0 , the address of the input parameters of P .
- An input vector \vec{I} , which associates a value to each input parameter, defines the initial value of \vec{M}_0 and hence \mathcal{M} .

In the bottom left corner, there is a small circular logo with the text "NPTEL" below it. In the bottom right corner, there is a small video inset showing a person's face.

What are the inputs to DART? Inputs to DART are the memory locations corresponding to the variables symbolic variables that actually correspond to the input. Input vector associates a value to each input parameters hence defines an initial value of memory. So, we assume as I told you for simplicity that a program has only conditional statements and only assignment statements. So, program execution will be a series of assignment statements and conditional statements written as $A \cup C$. Any number of them, which is written using the regular expression star and the program can either abnormally halt with an abort, or normally halt with a halt. So, read it like you would read a regular expression, they could be assignments, they could be conditions, star means any combinations of them, any number of them. This represents the program ending, abort represents the program ending abnormally, halt represents the program ending normally.


So, to further simplify and understand our model we can assume without loss of generality that program execution looks like this. It has a series of assignment statements followed by a condition, followed by another series of assignment statements followed by a condition and so on. Basically what it say is assignment statements and conditions could alternate. These α_i 's are assignment statements they belong to a star they - could be no α_i 's, C_i s are conditionals and then the program always ends with an abort or a halt s belongs to abort or a halt.

So, this is how a program execution looks like and whenever a program executes.

(Refer Slide Time: 27:07)

Symbolic evaluation of expressions

- DART maintains a **symbolic memory** \mathcal{S} that maps memory addresses to expressions.
- While the main DART algorithm runs, it will evaluate the symbolic path constraints using this algorithm and solve the path constraints to generate directed test cases.
- To start with, \mathcal{S} just maps each $m \in M_0$ to itself.
- The mapping \mathcal{S} is completed by evaluating expressions symbolically: algorithm in the next slide.
- Only linear path constraints can be solved. Hence, whenever the path constraints become non-linear, the algorithm just uses concrete values instead of symbolic values



There is one path, in the execution tree that the thing takes. So, I will stop here in the lecture. I will continue with how DART evaluates each of these expressions symbolically.

Thank you.

Software testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore


Lecture – 53
DART: Directed Automated Random Testing

Hello again, continuing from where we left in the last lecture on DART, I had ended with this explanation where we talked about the execution model of DART. I will begin by briefly recapping what we ended with in the last lecture.

(Refer Slide Time: 00:27)

Execution model of DART: Program Execution


- Let **C** be the set of conditional statements and **A** be the set of assignment statements in P .
- A **program execution** w is a finite sequence in $\text{Execs} := (A \cup C)^*(\text{abort}|\text{halt})$.
- They further simplify program execution and view w as being of the form $\alpha_1 c_1 \alpha_2 c_2 \dots c_k \alpha_{k+1} s$ where $\alpha_i \in A^*$, $c_i \in C$, and $s \in \{\text{abort}, \text{halt}\}$.
- An alternate view is to consider $\text{Execs}(P)$ is a tree, with assignment nodes having one successor, condition nodes have one or two successors and leaves being labeled by **abort** or **halt**.
- Each input vector results in an execution sequence, as a path in this tree.




We assume for simplicity sake, that every program has only assignment statements or an if condition which is represented as a conditional statement. Assignment statement is represented in the set A . Conditional statements are represented on the set C . So, program execution is a series of assignment statements and condition statements and always ends with an abort or halt statement. $\text{Execs}(P)$ can be thought of as an execution tree that consists of all program executions which we saw an example of when we did symbolic execution. Assignment nodes will have only one successor because there is an immediate next statement and that is only one. Conditional nodes will have a branch out based on whether the conditional is true or false and leaves will all be labeled by abort or halt.

(Refer Slide Time: 01:14)

Symbolic evaluation of expressions



- DART maintains a **symbolic memory** S that maps memory addresses to expressions.
- While the main DART algorithm runs, it will evaluate the symbolic path constraints using this algorithm and solve the path constraints to generate directed test cases.
- To start with, S just maps each $m \in M_0$ to itself.
- The mapping S is completed by evaluating expressions symbolically: algorithm in the next slide.
- Only linear path constraints can be solved. Hence, whenever the path constraints become non-linear, the algorithm just uses concrete values instead of symbolic values



So, now we will see how DART symbolically evaluates the expression. So, to symbolically evaluate the expression, DART maintains what is called a symbolic memory. What is a symbolic memory? Symbolic memory basically tells you for every variable which is a memory address in which the variable is stored. So, it maps memory addresses to expressions. For inputs, it is directly the name of the variable itself, but remember when we did that example for internal variables other variables, let us say if you had $z = x + y$, then you need to update the value of x by using an expression for x plus y as $x_0 + y_0$, where x_0 is the symbolic expression for x and y_0 is the symbolic expression for y . So, that is what we mean by a symbolic memory.

Symbolic memory denoted by S is a map from memory addresses to expressions. When the DART algorithm runs, it will evaluate symbolic path constraints using the algorithm when solved the path constraint to get directed generate test cases. Solving, it will give it to a third party constraint solver. To start with the initial the inputs will all be in S . The mapping of S is completed by evaluating expressions symbolically. So, for each kind of expression I will tell you how to symbolically evaluate the expression. Before we understand what to how to symbolically evaluate the expressions, fairly straightforward, whenever an actual variable comes in the expression you substitute the symbolic variable instead of the actual variable in the expression. But then ultimately we going to substitute, use these expressions in your path constraints to give it to a constraint solver, that is where the problem begins. As I told you constraint solvers cannot handle all kinds

of path constraints because the problem is un-decidable. In particular constraint solvers cannot handle path constraints that are not linear that is, path constraints of the form $5x^2 \neq 0, 3x^3 > 0$.


Any kind of non-linear expression; it is not good at handling. So, DART will remember this is one of the disadvantages of symbolic testing. So, whenever it encounters a non-linear path constraint, it drops it, in the sense that path constraints become non-linear algorithm, the DART algorithm; we will directly use the concrete values instead of symbolic values. That is how, it that is why DART keeps the concrete values set.

(Refer Slide Time: 03:50)

Symbolic evaluation of expressions

```

evaluate_symbolic(e, M, S) =
match e :
  case m :
    if m ∈ domain S then return S(m)
    else return M(m)
  case * (e', e'') :
    let f' = evaluate_symbolic(e', M, S);
    let f'' = evaluate_symbolic(e'', M, S);
    if not one of f' or f'' is a constant c then
      all_linear = 0
      return evaluate_concrete(e, M)
    if both f' and f'' are constants then
      return evaluate_concrete(e, M)
    if f' is a constant c then
      return * (f', c)
    else return * (c, f'')
```




So, here is how symbolic evaluation of expressions happen. There is a function called evaluate symbolic which symbolically evaluates an expression e in the context of a memory M and already available symbolic expressions capital S. So, what could be various kinds of expressions.

(Refer Slide Time: 04:15)

Execution model of DART: Symbolic variables

- **Memory** \mathcal{M} : a mapping from memory addresses m to say 32-bit words.
 - $+$ denotes updating of memory. For e.g., $\mathcal{M}' := \mathcal{M} + [m \mapsto v]$ is the same as \mathcal{M} except that $\mathcal{M}' = v$.
- **Symbolic variables** are identified by their addresses.
- In an expression, m denotes either a memory address or the symbolic variable identified by address m , depending on the context.
- A **symbolic expression** or just an expression, e is given by
 - m , c (a constant), $*(e, e')$ (multiplication), $\leq (e, e')$ (comparison), $\neg e'$ (negation), $*e'$ (pointer dereference) etc.
- Symbolic variables of an expression e are the set of addresses m that occur in it.
- Expressions have no side effects.



If you go back and see the various kinds of expressions could be just a variable a constant arithmetic expression a relational expression a logical expression or a pointer expression.

So, I have written only multiplication, but as I have told you in the last class, it represents all arithmetic operators this represent all relational operators, this represents all logical operators. So, we will do a case by case analysis of how it happens. So, if it is just a variable, then you directly assign it to the symbolic expression and you return the updated memory. If it is arithmetic expression which is represented by the $*(e, e')$, please read the star as standing for $+$, $-$, $/$, mod , $*$; all the arithmetic operators then it has 2 operands class an expression e and another expression e' , which first have to be evaluated symbolically themselves. So, e is evaluated symbolically, e' is, e' is evaluated symbolically, $e;$ is evaluated symbolically. This is a recursive call, evaluate symbolic calls itself.

But now if let us say this is a multiplication operator, right in this particular case. So, if both x and y are variables; let us f and f' involve variables, let us say f involves $5 \times y$, f' , f' involves $5 \times x$ and f'' involves y , then $5 \times y$ will become a non-linear expression and I just told you the non-linear expressions are difficult to solve by constraints solvers. So, DART has this condition, it says that if one of f and f' is a not a constant, not one of f or f' is a constant. If one of them is not a constant is I just told you now you will get a non-

linear expressions. So, if both are not constants or one of them is not to constant, then you say now I have an expression which is not linear. So, DART keeps a special flag which it calls all linear, sets that flag to 0. If it sets that flag to 0 which means what its intuitively saying the DART is encountered an expression that is non-linear.


So, DART will now do concrete evaluation, which is evaluate concrete of that expression, update the memory and go ahead, won't store the non-linear symbolic expression at all. Why will it not store, because this no point in storing it. DART knows if the constraint solvency that it is going to use cannot handle these expressions. If both f and f' are constants then there is no expression involving variables at all. So, DART does not have a choice, but to directly evaluate it concretely. That is what it does here; updates the memory and goes ahead. If one of them is a constant, then what it will do is it will correctly evaluate that expression and return.

(Refer Slide Time: 07:04)

Symbolic evaluation of expressions

```

case *  $e'$  :
  let  $f^h = \text{evaluate\_symbolic}(e', \mathcal{M}, \mathcal{S});$ 
  if  $f'$  is a constant  $c$  then
    if *  $c \in \text{domain } \mathcal{S}$  then return  $\mathcal{S} * c$ 
    else return  $\mathcal{M}(*c)$ 
  else  $\text{all\_locs\_definite} = 0$ 
    return  $\text{evaluate\_concrete}(e, \mathcal{M})$ 
  etc.
  
```



Now, this is the case, the rest of the operators that we saw an expressions where relational and logical operators the only other expressions was pointer related stuff. So, this is the second case, if its pointer, if it is the expression of the form $*e'$, then it has to first symbolically evaluate e' , so, it calls itself recursively. This is a continuation of this code on the slide. If f' is a constant, if it belongs to the domain of \mathcal{S} , then it returns whatever value that is. Otherwise it will return point 2 to the memory location. If it cannot determine either of this, then it will set another flag read it has all locations

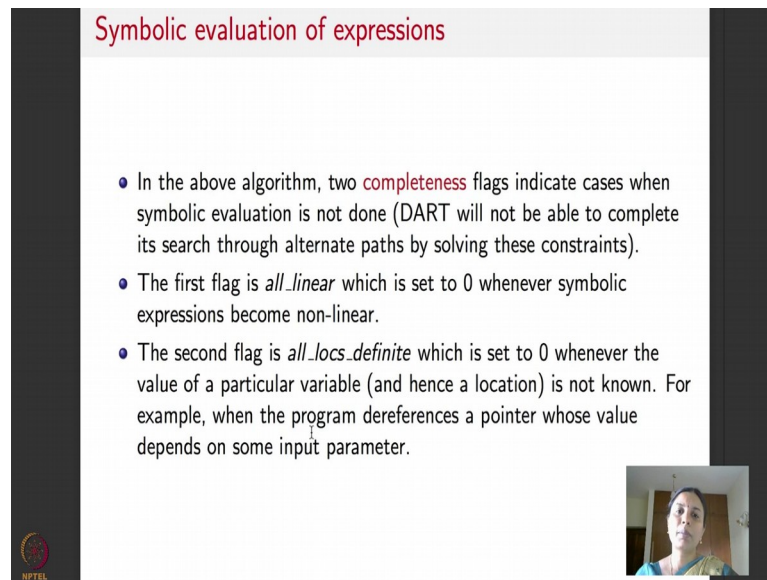
definite, which means it knows the locations of all the variables in the sense that if there is a point of pointer that it does not know, then it will say there is a location that I do not know which means it will set this flag all locations definite is to 0.

So, and then it will evaluate the concrete value and go ahead. So, what is this part then I am describing in. Remember just to recap what DART does? It takes a program, writes a driver, instruments it to first generate one random input runs the program on that random input. As the program is running on the random input, DART symbolically executes the program the symbolically execute the program. It has to symbolically evaluate all the expressions that is the part that I am describing here. So, symbolic evaluation is fairly straight forward it will keep substituting whatever expressions that it got a normal variables with symbolic variables that is what this recursive called does. There is one more thing that this pseudo code establishes. It says that if as I am executing, if I encounter an expression that is likely to be non-linear, that is if and symbolically evaluating e' and e'' and one of them after symbolically evaluating turn out to be, both turn out to be non not constants then I will say I have encountered linear expressions.

So, I will set this flag all near to 0, go back and substitute concrete values, I will not evaluated symbolically; not store it in the path constraint. Otherwise, I will do whatever normally evaluation is. Similarly here, when DART is handling pointer arithmetic, if it encounters a situation where in the memory address is not correctly known, then it will set to set flag which says all locations are not definitely known because here it does not evaluate all of them. So, they could see places where it does not know it, will go ahead and substitute concrete values otherwise it does normal symbolic execution. Why does it do the substitution of concrete values, because by doing this upfront, DART will ensure that if the path constraint that it collect is always solvable by the constraint solver that it is using.



So, this way it overcomes one of the disadvantages of symbolic testing.

(Refer Slide Time: 10:00)



Symbolic evaluation of expressions

- In the above algorithm, two **completeness** flags indicate cases when symbolic evaluation is not done (DART will not be able to complete its search through alternate paths by solving these constraints).
- The first flag is *all_linear* which is set to 0 whenever symbolic expressions become non-linear.
- The second flag is *all_locs_definite* which is set to 0 whenever the value of a particular variable (and hence a location) is not known. For example, when the program dereferences a pointer whose value depends on some input parameter.

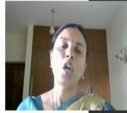
 

So, this is the part that I was trying to explain to you. In the algorithm that symbolically evaluate expressions; they were 2 completeness flags the DART kept. One flag was called all linear which is said to 0 whenever symbolic expressions become non-linear. The next flag, all locs definite or all locations definite is said to 0, whenever the value of a particular variable and hence the memory location of the variable is not known. When will such a situation arise? For example, you could consider a program which dereferences a pointer whose value depends on input parameters, then I will definitely not know the value of a particular variable right. So, these situations could quite commonly arise.

(Refer Slide Time: 10:43)

Test driver

```
run_DART() =
  all_linear, all_locs_definite, forcing_ok = 1, 1, 1
  repeat
    stack = {};  $\vec{I}$  = []; directed = 1
    while(directed) do
      try (directed, stack,  $\vec{I}$ ) =
        instrumented_program(stack,  $\vec{I}$ )
      catch any exception  $\rightarrow$ 
        if (forcing_ok)
          print "Bug found"
          exit()
        else forcing_ok = 1
    until all_linear  $\wedge$  all_locs_definite
```



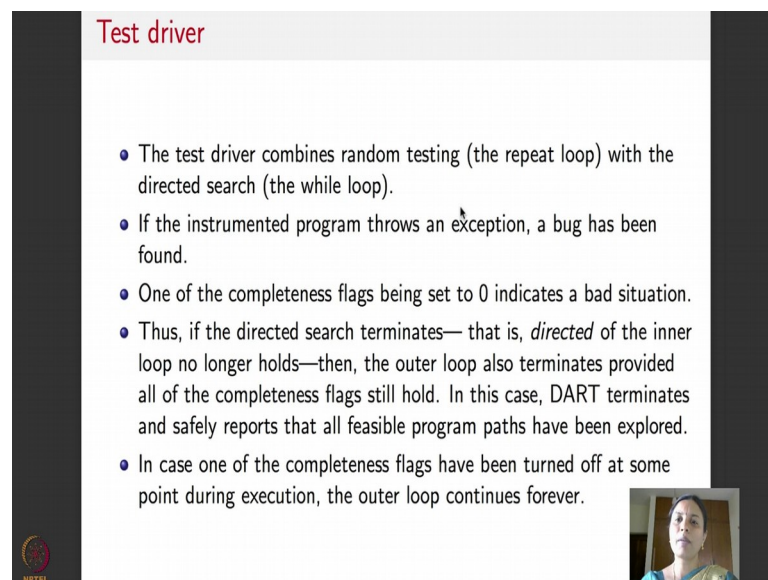
Now, let us look at the main algorithm of DART. How does the test driver of DART look like? So, this is the main program of DART, called the test driver or run DART. So, what does it do? First it initializes all these Boolean flags to 1. Read this as shorthand for saying, all linear, which is a Boolean variable to be 1 or true, all locations definite, which is another Boolean variable is assigned to 1 or true, forcing ok, which is another Boolean variable that I will explain. Now is also assigned to one or true. So, this is shorthand for 3 assignment statements which assigns 3 different Boolean variables each of them to the value true. Then DART goes into which mean loop. What is the main loop? The main loop is a repeat until loop, it is a repeat until this condition is true, which means what both the flags all linear and all locs definite stay as 1, then you keep on repeating this loop. The moment one of them become 0, you exit. Inside the loop what is DART do. DART keeps stack, we will see what the stack contains it initializes the stack to empty, i is the input vector the vector of all inputs it initializes it to empty.

It uses one more Boolean variable called directed which it initializes to one or true and then it enters a while loop. So, while this directed, directed stands for directed search. So, while I am doing directed search it has this try catch exception. So, what is try catch exception do? It at some point, this Boolean flag forcing is ever set to one, then DART is already found an error in the program, it will print this an exit. Otherwise DART, will go on instrumenting the program here. So, it will call this method called instrument program with the stack and i. So, is it clear please? What DART does? DART uses 3 Boolean

flags; all of them set to true and it goes into its main loop until all the path constraints that its encountering is linear and it knows exactly the locations of all the variables inside this loop DART will instrument the program and do a directed search.

At any point and time, if DART is found, the bug then DART will say I have found a bug it will print this message and come out.

(Refer Slide Time: 13:27)



Test driver

- The test driver combines random testing (the repeat loop) with the directed search (the while loop).
- If the instrumented program throws an exception, a bug has been found.
- One of the completeness flags being set to 0 indicates a bad situation.
- Thus, if the directed search terminates— that is, *directed* of the inner loop no longer holds—then, the outer loop also terminates provided all of the completeness flags still hold. In this case, DART terminates and safely reports that all feasible program paths have been explored.
- In case one of the completeness flags have been turned off at some point during execution, the outer loop continues forever.

NPTTEL

13

So, this is how the test driver looks like. It combines random testing which is this repeat loop with directed search which is in a while loop. If the instrumented program throws an exception, a bug has been found. So, DART will print that message and exit you one of the completeness flag is being said to 0 means, what DART is encountered a bad situation, it is encountered a non-linear constraint, it is encountered place where path memory of a particular variable is not known, it is encountered some bad situation.


So, if the directed search terminates then the directed variable, which is this variable of this inner loop no longer holds, then the outer loop will also terminate provided the completeness flag still hold. That is clear from this code, I hope. In this case, DART terminates and safely reports that all the feasible program paths have been explored. But in case, one of the completeness flags have been turned off which is this, all linear or all locations definite, the outer loop will continue forever. So, they could be 3 options; DART will explore all possible program paths successfully, DART will find an error and stop saying that I found an error or DART can run forever. That is not too surprising

because it is a program analysis tools and the problem that is trying to handle which is exploring all program paths is in general and undeniable problem.

(Refer Slide Time: 14:33)

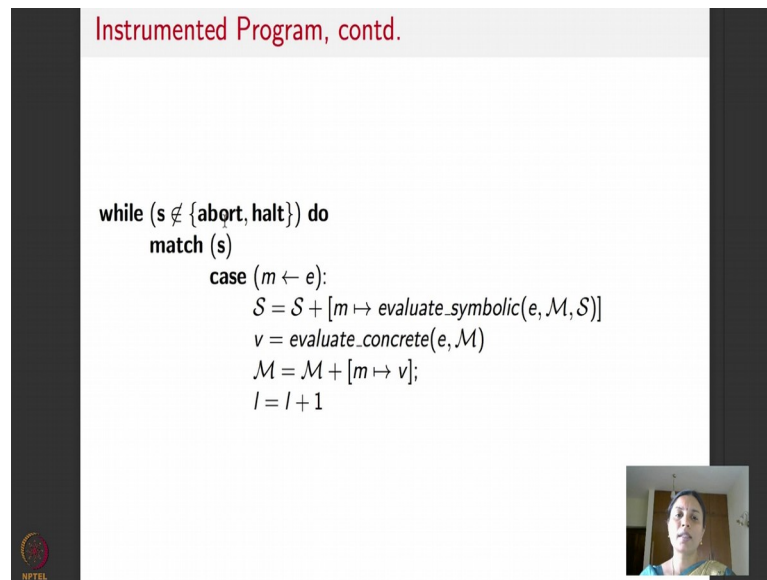
Instrumented Program

```
instrumented_program(stack,  $\vec{l}$ ) =  
// Random initialization of uninitialized i/p parameters in  $\vec{M}_0$   
for each input  $x$  with  $\vec{l}[x]$  undefined do  
     $\vec{l}[x] = \text{random}()$   
Initialize memory  $\mathcal{M}$  from  $\vec{M}_0$  and  $\vec{l}$   
// Set up symbolic memory and prepare execution  
 $S = [m \mapsto m \mid m \in \vec{M}_0]$   
 $l = l_0$  // Initial program counter in  $P$   
 $k = 0$  // No. of conditionals executed  
// Now invoke  $P$  intertwined with symbolic calculations  
 $s = \text{statement\_at}(l, \mathcal{M})$ 
```



So, we will now look at this code. What does the code corresponding to instrumented program in the main test driver of DART looks like. So, here how instrumented program code looks like. It takes a stack and a vector of inputs. So, first it does a random initialization of uninitialized input parameters in the memory locations. So, it says for each input x with a $\vec{l}[x]$ uninitialized do random. This is the initial random text vector that DART generates. It initializes memory to this value that it is found, it sets up the symbolic memory which is S , sets the initial program goes and says, I have not yet encountered any path constraint and begins at the first statement.

(Refer Slide Time: 15:27)



Instrumented Program, contd.

```
while (s ∉ {abort, halt}) do
  match (s)
  case (m ← e):
    S = S + [m ↦ evaluate_symbolic(e, M, S)]
    v = evaluate_concrete(e, M)
    M = M + [m ↦ v];
    l = l + 1
```

And as long as it is not encountering an abort or halt, DART will encounter either an assignment statement or it will encounter a condition statement because we have assumed that program execution is a series of, I will go back to back that slide first.



We have assume that the program execution is a series of assignments and conditions ending with an abort or halt. So, that is what this instrumented program does. So, it sets up the memory, it sets its stat counter to be 0 and then it begins. So, which is as long as statements are not abort or halt, it is either an assignment statement or a condition statement. So, there is a switch case like. So, S is match to an assignment statement, what will it do? It is already has a symbolic memory uploaded. Here is a new assignment statement that it is encountered. For example, this could be the case there which says and $z = x + y$. So, the memory location for z which is M needs to be updated with the expression x0 for $x_0 + y_0$. So, the memory location for M is updated after evaluating symbolically the expression e with the memory location M and with the symbolic set of expressions S.

It also needs to evaluate the expression e concretely because you do not know when in the future it will have to substitute. So, it will evaluate symbolically, it will evaluate concretely. It will update the memory location and it says I finished with this statement. So, we will update the program counter to one.

(Refer Slide Time: 16:56)

Instrumented Program, contd.

```
while (s  $\notin$  {abort, halt}) do
  match (s)
    case (if (e) then goto l') :
      b = evaluate_concrete(e, M)
      c = evaluate_symbolic(e, M, S)
      if b then
        path_constraint = path_constraint ^ (c)
        stack = compare_and_update_stack(1, k, stack)
        l = l'
      else
        path_constraint = path_constraint ^ ( $\neg$ c)
        stack = compare_and_update_stack(0, k, stack)
        l = l + 1
        k = k + 1
      s = statement_at(l, M) // End of while loop
```



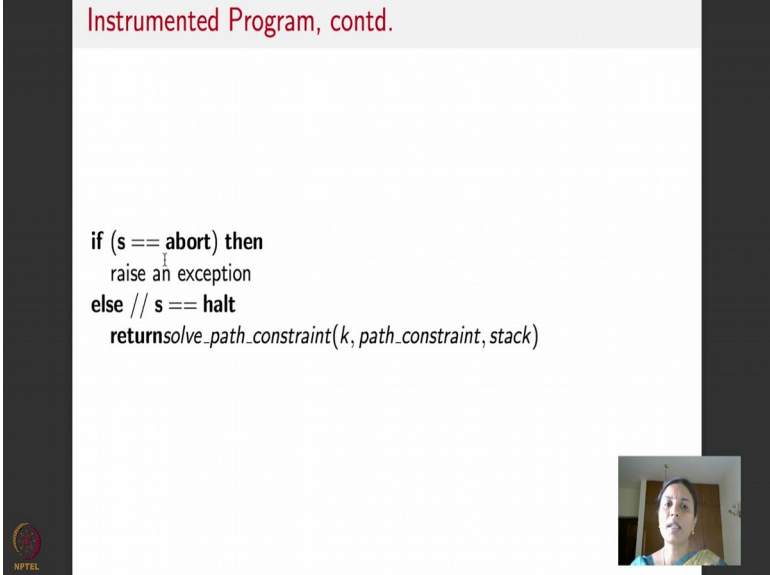
So, let us say DART encounters a condition statement. If the match happens to be a condition. So, it says if e then go to l', then what it will do is that it will evaluate e symbolically and concretely. Let us say b is the symbolic, result of concrete evaluation of e, C is the result of symbolic evaluation of e. If b is true then the then path is taken by the program if b is false, then the else path is taken by the program.

So, if b is true, then it already has a path constraint which it has initialized here it will go ahead and the condition that it got by symbolically evaluating it to the path constraint. Read this as whatever the path constraint that was earlier now C is added to it with an AND and then, it will now put this in this stack saying, I have one more path constraint to update, I found on the way. The program, took a branch, here is a path constraint the program took a then branch, so, I am putting a one here and adding this constraint to the stack and it will change the statement to whichever that it has to go to, l'. But suppose the condition evaluates to falls then to the path constraint it will add negation of that condition. This is path constraint ended with negation of C, it will update it stack by saying the program took the else branch with the 0 and add this to the stack and change the statement number because it is like skipping.

So, statement number now becomes l plus 1 and it will go ahead and do it, is this clear. So, basically this part what it does is that it grows the path constraint one at a time, one at a time and what is the stack do? Stack keeps track of which is the path constraint that

was added the latest because that is the path constraint the DART is going to flip to be able to get the next execution of the program. That is what DART does here right, it will flip it in the in the main thing when it does the next execution of the program.

(Refer Slide Time: 18:55)



Instrumented Program, contd.



```
if (s == abort) then
    raise an exception
else // s == halt
    returnsolve_path_constraint(k, path_constraint, stack)
```

So, if S is an abort or halt, if S is an abort then DART raises an exception, if S is a halt then it just re-directly calls the constraint solver with the path constraint that it is collected as it takes it from the stack. Now in this it used this routine called compare and update stack. What is that do, it tells you how to update the stack here is the code for that.

(Refer Slide Time: 19:06)

Compare and update stack routine

```
compare_and_update_stack(branch, k, stack) =  
if k < |stack| then  
  if stack[k].branch ≠ branch then  
    forcing_ok = 0  
    raise an exception  
  else if k = |stack| - 1 then  
    stack[k].branch = branch  
    stack[k].done = 1  
  else stack = stack[(branch, 0)]  
return stack
```





So, stack has a maximum capacity this part says as long as if not reached the maximum capacity if the stack branch is not equal to branch then there is a problem four set forcing to 0 and raise an exception. Where is this Boolean variable forcing ok, it was here. So, if forcing say ever set, then you can say bug is found l's is stack. This; like this then you go ahead and normally update the stack branch, you say you to remove it from the stack profit off and then add this. Is this clear? This normally just the normal stack operations that you are doing right solve path constraint which is what it calls here right solve path constraint what is that look like?

(Refer Slide Time: 19:58)

Solve path constraint routine

```
solve_path_constraint(k_try, path_constraint, stack) =  
  let j be the smallest number such that  
    for all h with  $-1 \leq j < h < k_{try}$ , stack[h].done = 1  
  if j = -1 then  
    return (0, -, -) // Directed search is over  
  else  
    path_constraint[j] = ¬(path_constraint[j])  
    stack[j].branch = ¬stack[j].branch  
    if (path_constraint[0, ..., j] has a solution  $\vec{l}$ ) then  
      return(1, stack[0..j],  $\vec{l} + \vec{l}$ )  
    else  
      solve_path_constraint(j, path_constraint, stack)
```




That will basically call a constraint solver which is third party constraints solver. So, what it does is that it takes the latest path constraint from latest thing from the stack and its says if j is -1, then I finish my search it terminated normally. Otherwise it says do take this whatever path constraint that you found at the top of the stack negate that path constraint added to the branch now you call it again. So, that it explores the other search, this is the part that forces start to do the directed search. This is an explanation of what the instrumented program does. Each run of the instrumented program, the first one does a random search.

(Refer Slide Time: 20:32)

Instrumented program

- Each run of the instrumented program (except the first) is executed with the help of a record of conditional statements executed in the previous run.
- For each conditional, we record a *branch* value and a *done* value.
- *branch* value is 1 when the **then** branch is taken and 0 otherwise.
- *done* value is 0 only when one branch of the conditional has executed in prior runs (with the same history up to the branch point) and is 1 otherwise.
- This information associated with each conditional statement of the last execution path is stored in a list variable called *stack*, kept in a file between executions. For i , $0 \leq i < |stack|$, $stack[i] = (stack[i].branch, stack[i].done)$ is thus the record corresponding to the $(i + 1)$ th conditional executed.




So, except for the first one, each run of the instrumented program, how is it executed? You have a stack of all the conditions that you encountered in the path, in the previous run. So, we record a branch value and a done value. Branch value is 1, as I told you when the then part is true or the condition is true and branch value is 0 if the condition is false. Done value 0 only when the branch of the condition is executed in the prior runs which means I have explored this part already, so then you said the fully done with this program constraint. Information associated with each conditional statement of the last execution path is stored in a variable called stack, which is what we have been looking at and that is kept in a file that shared between executions and then at any point in time, the whatever is available at the top of the stack is pulled, flipped and that is the alternate path the DART tries to take.

(Refer Slide Time: 21:40)

Instrumented program

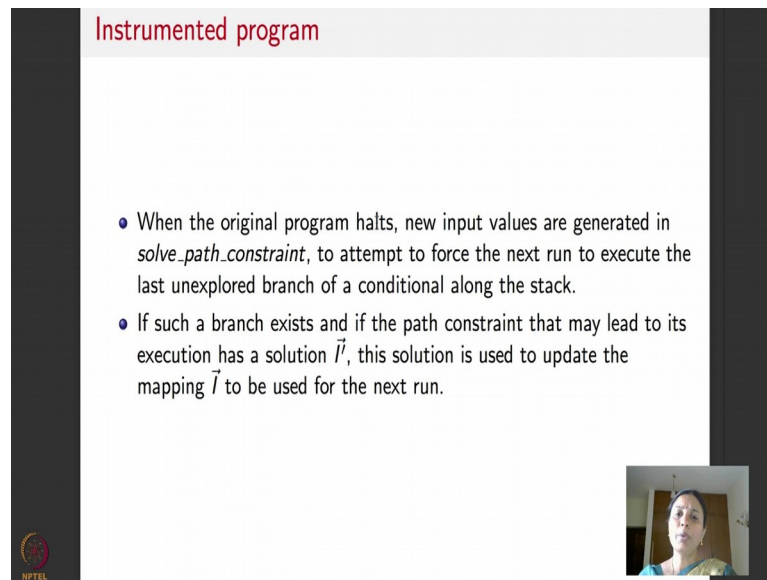
- The instrumented program executes the original program with interleaved gathering of symbolic constraints.
- At each conditional statement, it also checks by calling *compare_and_update_stack*, whether the current execution path matches the one predicted at the end of the previous execution and represented in *stack* passed between runs.
- If it ever happens that a prediction of the outcome of a conditional is not fulfilled, then the flag *forcing_ok* is set to 0 and an exception is raised to restart *run_DART* with a fresh random input vector.
- Note: setting *forcing_ok* to 0 can only be due to a previous incompleteness in DART's directed search, which was then detected and resulted in setting at least one of the completeness flags to 0. In other words, the invariant $all_linear \wedge all_locs_definite \Rightarrow forcing_ok$ holds.



So, if it explored both the options, then it such the done value to 0, removes it from the stack and takes the next available condition. Instrumented program what is it do? It basically executes the original program interleaved with a gathering of symbolic constraints. At each conditional statement, it checks the, it calls this compare and update stack routine, it checks whether the current execution path matches the one that it predicted at the end of the earlier execution and if it is, then he does not repeat, but if it not then it explores this new execution path. But if some problem happens then it sets the flag forcing to 0 and basically then it will restart this run DART which is the main test driver with another fresh randomly generated input.



When will the sourcing be said to 0 that can be only due to a previous incompleteness and darts directed search, it went along a path that it could not complete, so it will start all over again.

(Refer Slide Time: 22:29)



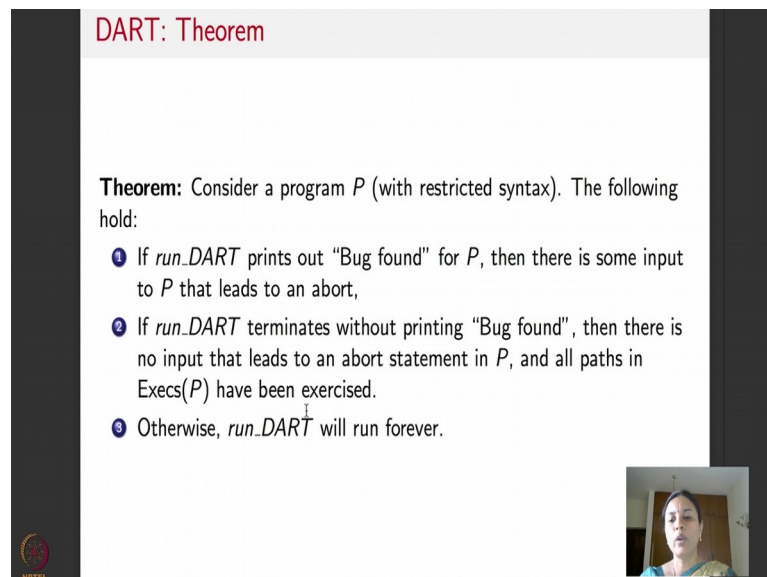
Instrumented program

- When the original program halts, new input values are generated in *solve_path_constraint*, to attempt to force the next run to execute the last unexplored branch of a conditional along the stack.
- If such a branch exists and if the path constraint that may lead to its execution has a solution \vec{I}' , this solution is used to update the mapping \vec{I} to be used for the next run.

So, when the original program halts, new input values are generated from the solve path constraint routine that will attempt to force the next run to execute, the last unexplored branch of the stack such a branch exists, then it is a explored. Otherwise, it removes it from the stack can goes to the next path constraint that it was first.



(Refer Slide Time: 22:48)



DART: Theorem

Theorem: Consider a program P (with restricted syntax). The following hold:

- 1 If *run_DART* prints out "Bug found" for P , then there is some input to P that leads to an abort,
- 2 If *run_DART* terminates without printing "Bug found", then there is no input that leads to an abort statement in P , and all paths in $\text{Execs}(P)$ have been exercised.
- 3 Otherwise, *run_DART* will run forever.

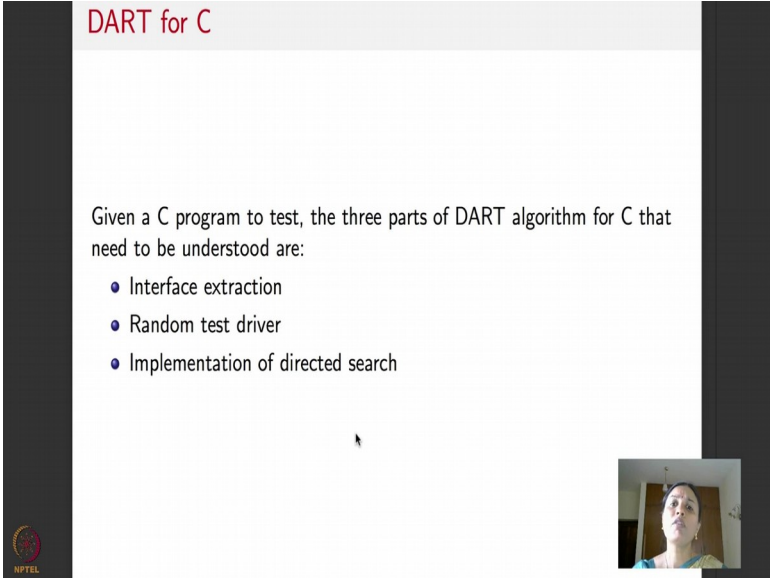
 

So, I hope the code for DART is clear. So, the main code for DART is the test driver which does one random search followed by a program instrumentation which symbolically executes a program, collects the path constraint and does a directed search.

This was the code for that which tells you how to symbolically update and execute a program. It needs a stack to be able to keep track of the path constraints, this is a update, this is the routine method that maintains the stack, this is the routine that calls the constraint solver and DART could as I told you have 3 outcomes.

So, here is the main theorem that talks about the correctness of DART. If DART runs, if DART at any point and time says this print bug found, then there is some input to be that leads to an abort state, it is really found an error. If run DART terminates without printing bug found, then which means or DART is explored all the execution paths of P. Otherwise run DART will run forever somebody has to manually go on about it. The third part is an undesirable behavior of DART, which we cannot avoid because the underlying problem is undesirable

(Refer Slide Time: 23:59)



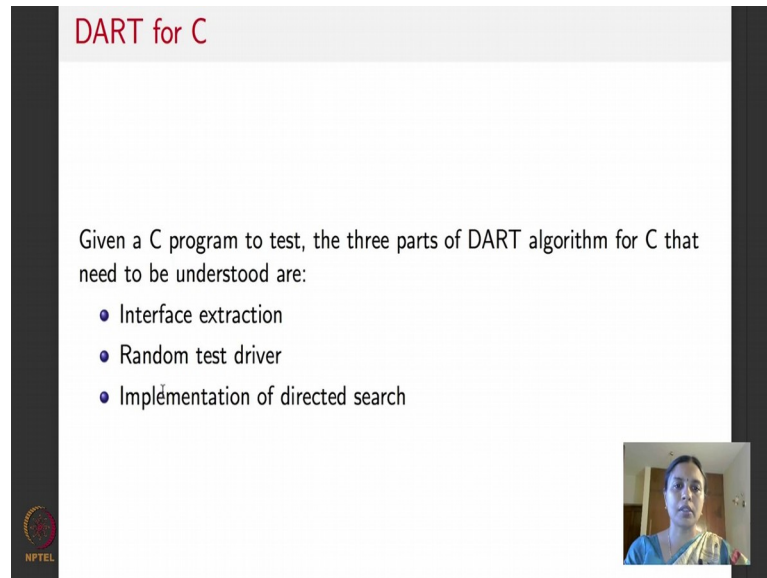
What will do next time is I will explain how DART works for C by using an example and tell you how these interface extraction test driver extraction undirected search implementation actually happens through an example. So, we will stop with this one now.

Thank you.

Software testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 54
DART: Directed Automated Random Testing

(Refer Slide Time: 00:13)



DART for C

Given a C program to test, the three parts of DART algorithm for C that need to be understood are:

- Interface extraction
- Random test driver
- Implementation of directed search

The slide features a dark grey header with the title 'DART for C' in red. The main content is on a white background. In the bottom left corner, there is a small NPTEL logo. In the bottom right corner, there is a small video inset showing a person speaking.

Welcome back, I am going to finish lecturing about DART with this lecture. Just to recap what DART is, DART stands for directed automated random testing, an independent standalone unit testing tool for a C program or Java program and it basically does concolic execution, a mix of concrete and symbolic execution. What are the steps that DART follows? DART begins by generating a random test input for the given C program. It does enough work to figure out how to generate the random test input such that the it is a correct format in test input for the C program, runs the C program on this randomly generated inputs and collects a few pieces of information.

What does it collect? It collects the details about the path that is randomly generated test input takes the C program on, it collects all the constraints that the that were encountered along the path puts them in a stack. Along with that whenever a particular constrain that it is encountering along a path, turns out to be non-linear because it cannot be solved by a constraint solver, DART also keeps the concrete values substitutes the concrete value,

and raises a flag which says that I am not able to symbolically execute this particular path.


After it is generated successfully the path constrain for a randomly generated test vector, DART systematically explores neighboring paths that the program would have taken. By taking one constrain at a time that it encountered on this random test vector, which, it kept these constraints in a stack, if you remember, takes it out of the stack flips the constrain, which means sorry negates the constraint and explore the neighboring program path. Once its explored a particular constraint fully, it goes on to the next up constraint on the program which is below this in the stack, again systematically flips it keeps doing this. When its able to successfully terminate doing this then DART has done what is called a reasonably thorough directed search of a program. But at some point if it encounters an abort statement, then DART has found a bug. So, it will report saying it is found that and stock, but at some point one of the flags which is all linear, all locations definite turned to be set to be 0 by dart; that means, DART is abnormally terminated.

Another option could be there on a non terminating program DART could run forever. Now we understand the algorithm for DART, I will tell you how for a program written in a programming language C, DART will go about interface extraction which it needs to do because it has to figure out what is an input to a program, and how to give the program its input, how does it actually do the random test driver, which generates the random input and how the directed search is implemented.

(Refer Slide Time: 03:02).

AC-controller Example in C

```
int is_room_hot = 0;
int is_door_closed=0;
int ac=0;
/* initially room is not hot, door is open and AC is off */
void ac.controller(int message) {
    if (message == 0) is_room_hot=1;
    if (message == 1) is_room_hot=0;
    if (message == 2) {
        is_door_closed=0;
        ac=0; }
    if (message == 3) {
        is_door_closed=1;
        if (is_room_hot) ac = 1 ; }
    if (is_room_hot && is_door_closed && !ac)
        abort(); /* check correctness */
}
```




So, we will take a small example. Here is a small example of a controller that controls the air conditioning this program is written in C. So, only again as always a program segment is given here. So, is room hot is an integer variable which is initialized to 0, which means the room is not hot, is door closed is another integer variable initialized to 0, which means that the door is open; ac 0 means the AC is off.

So, initially the room is not hot door is not closed or door is open and the ac is off. AC controller takes as an argument or message which is an integer variable. So, message could take values 0 1 2 or 3. If message is 0, then what it means is that it sets room hot to one, which means its starts heating up the room. If message is 1 then its starts cooling the room. If message is 2 then what it does is that its closes the door and puts off the AC. If message is 2 then it opens the door and puts off the AC. If a message is 3 then it closes the door and if the room is still hot it switches on the AC. And if the room is hot the door is closed and the AC is not on, there is an error in the program; because the doors closed AC is not on the room is very hot which means the controller is not working. So, it goes into a state for abort this is like an error state for the AC control.


So, this is the small program which tries to take a message may be through a reader from a user who has AC as part of the home control system, and tries to do some simple command.

(Refer Slide Time: 04:44)

DART for C: Interface extraction




- Apart from the inputs that are initialized, DART first identifies the external interfaces through which the program can obtain inputs via uninitialized memory locations \vec{M}_0 .
- For C, the external interfaces of a program are:
 - its external variables and external functions (reported as "undefined reference" at the time of compilation of the program), and
 - the arguments of a user-specified *top-level function*, which is a function of the program called to start its execution.
- Such external interfaces are obtained by DART using static parsing of the source code.



We will see how DART will test such a program and successfully ensure that it reaches this abort statement here. So, now, DART for C, the first step as we told you is to do interface extraction. How does it do that? Now apart from inputs that are initialized by a program, there are there could be other inputs to a program, DART first works on identifying them. How does it identify? It has to identify the external interfaces through which the program obtains its inputs and these are typically from uninitialized memory locations, because these are inputs that are not initialized. For a program written in C typical interfaces could be its external variables, its external functions or it could be arguments of a user specified top level function, which is the function of the program to call to start it is a execution, it could be any of these kinds. These external interfaces are obtained by DART by statically passing the code of the C program.

(Refer Slide Time: 05:32)



DART for C: Interface extraction

- Inputs to a C program are defined as memory locations which are dynamically initialized at runtime through the static external interface.
- DART for C supports two ways of mapping inputs to memory addresses:
 - Multiple inputs can be mapped to an address m and these are obtained by successively reading m during different successive calls to the top-level function.
 - Same input can be mapped to different addresses in different executions, for example, when the input is provided through an address dynamically allocated with `malloc()`.
- For each external interface, we determine the type of the input that can be passed to the program via that interface.
- In C, a type is defined recursively as either a basic type (int, float, char, enum etc.), a struct type composed of one or more fields of other types, an array of another type or a pointer to another type.

Now, after it passes the code how does it extract the interfaces? The interface extraction happens as follows. Inputs to the C program; how are they defined? The inputs are basically variables in the C program. So, all of them will have memory locations. These memory locations are dynamically initialized, at run time when the program is running through a static external interface the dart creates.



So, how does dart do that? Its suppose two different ways of mapping inputs to memory addresses. First way is multiple inputs can be mapped to one memory address m and all these inputs are obtained by successively reading m during different calls successively to the top level function. This is standard way in which C works or the same input can be mapped to different addresses because program could take different executions. For example, the same input could be provided through an address that is dynamically allocated with a command like `malloc` anything could happen. For each of these kind of external interfaces DART determines the type of the input that can be pass through the program via that interface. So, that it can be generate a random number compatible with that type. For C, type could be a basic type that is supported by C like for example, C supports integer type, floating point type, character enum and so on. It could be a struct type composed of one or more field of other types, it could be an array of another type a pointer, it could be any of the standard types that are supported by C.

(Refer Slide Time: 07:06)

DART for C: Interface extraction

DART distinguishes three kinds of C functions:

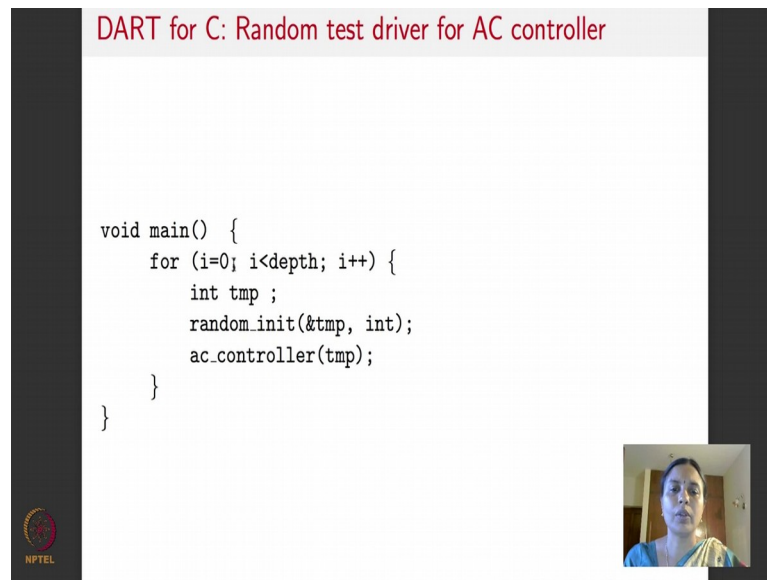
- **Program functions** are functions defined in the program.
- **External functions** are functions controlled by the environment and hence part of the external interface of the program; they can non-deterministically return any value of their specified return type.
- **Library functions** are functions not defined in the program but controlled by the program, and hence considered part of it, e.g. functions defined in the standard C library. These functions are treated as unknown but deterministic black-boxes which DART cannot instrument or analyze.



Now, what happens after this is yeah, so, DART while it does interface extraction it distinguishes three different kinds of functions, there are program functions which are basically functions or procedures that are defined as a part of the program itself, they could be external functions which have functions controlled by the environment. They are part of the external interface of the program, but they are not a part of the co program so, we cannot presume many things about the value that they would return.

They could non deterministically return any value of the compatible type or in a programming language like C you could have several library functions, which have functions not defined in the program, but they are used and controlled by the program. So, they considered a part of it. But because they are library functions DART does not have direct access to the code of these functions. So, dart still treats these functions as unknown functions, but they are deterministic. Deterministic in the sense that I know what a function does, so, it leads like a black box. So, DART really does not go into the function code to instrument it or analyze it. It substitutes a value of the compatible type and moves on. So, for this AC controller program that I showed you in this slide, what are the inputs? Inputs is basically this message which tells you what is the state is that the room should be maintained, in terms of the room being hot or cold, door being open or closed and ac being on or off. So, that is what happens in the external interface here.

(Refer Slide Time: 08:31)



DART for C: Random test driver for AC controller

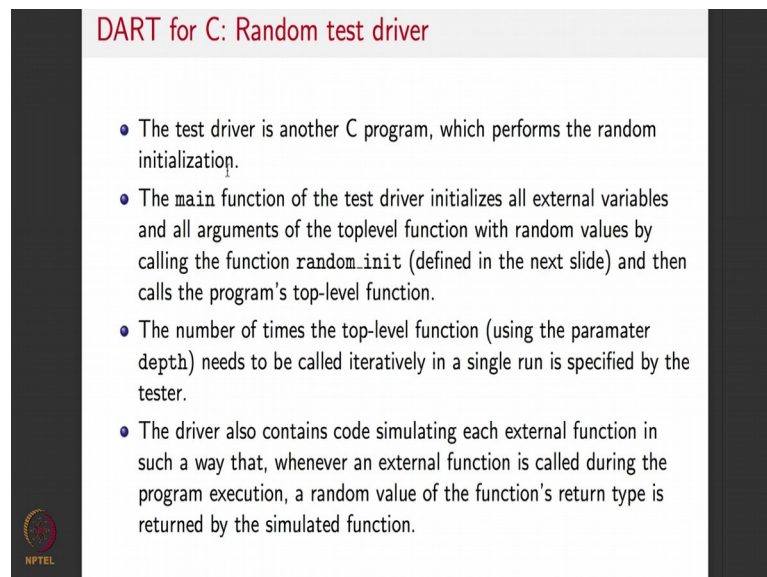
```
void main() {  
    for (i=0; i<depth; i++) {  
        int tmp ;  
        random_init(&tmp, int);  
        ac_controller(tmp);  
    }  
}
```

NPTEL

A small video inset in the bottom right corner shows a person speaking.

So, it says. So, this is the random test driver. So, it calls a function main, and it says for i is equal to 0 to, up to the maximum depth, you have a local variable called tmp and you randomly initialize tmp with this int and call ac controller with this tmp. So, what is this random in it function do? That is what we will try to understand now.

(Refer Slide Time: 08:56)



DART for C: Random test driver

- The test driver is another C program, which performs the random initialization.
- The main function of the test driver initializes all external variables and all arguments of the toplevel function with random values by calling the function random_init (defined in the next slide) and then calls the program's top-level function.
- The number of times the top-level function (using the parameter depth) needs to be called iteratively in a single run is specified by the tester.
- The driver also contains code simulating each external function in such a way that, whenever an external function is called during the program execution, a random value of the function's return type is returned by the simulated function.

NPTEL

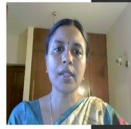

So, the test driver is another C program which performs the random initialization, that is this function random in it. The main function of the test driver initializes all the external variables and all the arguments to the top level function by calling this function random

init. That what, this main function basically calls the random in it and then calls the programs top level function. So, we will see random in it in the next slide. The number of times the top level function is called, that is specified by tester. Like for example, this depth no, how many times to call this random initialization you run it once you run it twice that is up to as, may be its one dart attempted and was not successful. So, you might want to run it again. So, this depth value is determined by the tester. That basically calls the random init functions as many times as it is needed. This driver also contains code for simulating each external function, in such that whenever an external function is called during a program execution, random value of the function return type is returned by the simulated function.

(Refer Slide Time: 10:06)

random_init: Procedure for randomly initializing C variables of any type

```
random_init(m, type) {  
    if (type == pointer to type2) {  
        if (fair coin toss == head) {  
            *m = NULL;  
        }  
        else {  
            *m = malloc(sizeof(type));  
            random_init(*m, type2);  
        }  
    }  
}
```





So, here is how the random code looks like, random init takes an argument m of a particular type. We do not know what a type could be, it could be anything that is supported by c it could be any type. So, I have just mentioned it as type left it like this.

So, now we will see what it is. So, if type is actually a pointer to type 2, then what you do? This is basically a random number generator. So, you toss a fair coin and if it turns out to be head then you say its null otherwise you allot chunk of memory of the size of type whatever that size is, and you randomly initialize that memory to be something of this type 2. I hope this is clear right, it is just the normal random initialization procedure.

(Refer Slide Time: 10:51)


random_init: Procedure for randomly initializing C variables of any type

```
} else if (type == struct) {  
    for all fields f in struct  
        random_init(&(m → f), sizeof(f));  
} else if (type == array[n] of type3) {  
    for (int i=0; i<n; i++)  
        random_init((m+i), type3);  
} else if (type == basic type) {  
    *m = random_bits(sizeof(type));  
}  
}
```



But let say if the type, so, this is the type which is a pointer to another type called type 2. So, you do this randomly allotted memory of the size and initialize it with this one. But if it is a struct type then what do you do for all fields of f in that struct, you randomly initialize each value for the field of the appropriate type. If it is an array type let say of some other type, if it is an array of size n of some other type, then you randomly initialize the value of the array of that same type 3. But if it is a basic type then you generate random bits of that same type. So, this basically tells you that random initialization function does a correct random initialization based on whether the type being a pointer, type could be a struct, type could be an array, type could be one of the basic types. Whatever it is it correctly allots a random input of that size, that type, that is compatible to the correct size.

(Refer Slide Time: 11:49)



DART for C: `random_init`

- `random_init` takes a memory location `m` and the type of value to be stored at `m` as arguments, and initializes `m` randomly depending on its type.
- If `m` stores a value of basic type, its value `*m` is initialized with the auxiliary procedure `random_n_bits` which returns `n` random bits, `n` being its argument.
- If its type is a pointer, location `m` is randomly initialized with either the value `NULL` (with probability 0.5) or with the address of newly allocated memory location, whose value is in turn initialized according to its type following the same recursive rules.
- If type is struct or array, every sub-element is initialized recursively in the same way.



Now, after, what is `random_init` do? It takes a memory location `m` and a type of the value to be stored at `m` as arguments, and initializes `m` randomly depending on its type. That is what is return here. Takes a memory location `m` along with its type it initializes them randomly of this type. If `m` is a value of basic type, then the pointed to `m` is initialized with auxiliary procedure `random bits`, which is this. Generate some bits of a compatible type. If the type is a pointer, then it is a randomly initialized either with an value null or with address of a newly allocated memory location, this is the one, which calls itself again because it is a recursive call and there is fair coin toss. So, the chances here are 50 50. If the type is struct or array every sub element is initialized recursively the same way. That is what is written here. If the type is of struct or the type is a array then you call the same `random_init` to initialize it till you get to the basic type.

(Refer Slide Time: 12:51)

DART for C: Random test driver

- For each external variable or argument to top-level function, DART generates a call to `random_init(&v, sizeof(v))` in the function `main` of the test driver before calling the top-level function.
- If the C program being tested calls an external function, say `return_type some_fun()`, then the test driver generated by DART will include a definition for this function:

```
return_type some_fun() {  
    return_type tmp;  
    random_init(&tmp, return_type);  
    return tmp;  
}
```
- Once the test driver has been generated, it can be combined with the C program being tested to form a self-executable program.





Now, let us look at the go back to the test driver. So, for each external variable argument to top level function, DART generates a call to the random init in the function of the main test driver before calling the top level function, that is the part that we saw till now. If the C program being tested calls an external function, lets say of something like this let say some function call it some name, some fun of some return type, then the test driver generated by DART will also include a definition for that function.

So, it will say this is the return type of this function called some function, inside that the return type there is a variable called tmp which is of type return type, and the random init function can be called to initialize tmp and return it. So, what it basically does is that if there is a particular function, what DART does is basically creates a stub for that function, and returns a type, that is variable that is compatible type to the actual variable that the function would return. So, once this test driver has been generated, now we can combine it with a C program to form an self executable program that is ready for instrumentation.

(Refer Slide Time: 14:03)

Instrumented Program

```
instrumented_program(stack,  $\vec{l}$ ) =  
  // Random initialization of uninitialized i/p parameters in  $\vec{M}_0$   
  for each input  $x$  with  $\vec{l}[x]$  undefined do  
     $\vec{l}[x] = \text{random}()$   
  Initialize memory  $\mathcal{M}$  from  $\vec{M}_0$  and  $\vec{l}$   
  // Set up symbolic memory and prepare execution  
   $\mathcal{S} = [m \mapsto m | m \in \vec{M}_0]$   
   $l = l_0$  // Initial program counter in  $P$   
   $k = 0$  // No. of conditionals executed  
  // Now invoke  $P$  intertwined with symbolic calculations  
   $s = \text{statement\_at}(l, \mathcal{M})$ 
```

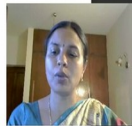



Once it is ready for instrumentation then I will go back for a minutes to the code that I showed you last time which is basically this.

(Refer Slide Time: 14:16)

Instrumented Program, contd.

```
while ( $s \notin \{\text{abort}, \text{halt}\}$ ) do  
  match ( $s$ )  
    case (if ( $e$ ) then goto  $l'$ ) :  
       $b = \text{evaluate\_concrete}(e, \mathcal{M})$   
       $c = \text{evaluate\_symbolic}(e, \mathcal{M}, \mathcal{S})$   
      if  $b$  then  
         $\text{path\_constraint} = \text{path\_constraint} \wedge \langle c \rangle$   
         $\text{stack} = \text{compare\_and\_update\_stack}(1, k, \text{stack})$   
         $l = l'$   
      else  
         $\text{path\_constraint} = \text{path\_constraint} \wedge \neg \langle c \rangle$   
         $\text{stack} = \text{compare\_and\_update\_stack}(0, k, \text{stack})$   
         $l = l + 1$   
         $k = k + 1$   
   $s = \text{statement\_at}(l, \mathcal{M})$  // End of while loop
```





So, it sets up symbolic memory it sets up the stack updates the symbolic memory keeps going updates the stack, and does all, the solving the path constraint and then runs the program.

(Refer Slide Time: 14:20)

Solve path constraint routine

```
solve_path_constraint( $k_{try}$ , path_constraint, stack) =  
  let  $j$  be the smallest number such that  
    for all  $h$  with  $-1 \leq j < h < k_{try}$ ,  $stack[h].done = 1$   
  if  $j = -1$  then  
    return (0, -, -) // Directed search is over  
  else  
    path_constraint[j] =  $\neg(path\_constraint[j])$   
    stack[j].branch =  $\neg stack[j].branch$   
    if (path_constraint[0, ..., j] has a solution  $\vec{I}'$ ) then  
      return(1, stack[0..j],  $\vec{I} + \vec{I}'$ )  
    else  
      solve_path_constraint(j, path_constraint, stack)
```



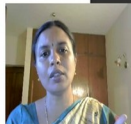

Once its runs the program it can return in it can terminate in any of these three values, it can run forever which is bad.

(Refer Slide Time: 14:24)

DART: Theorem

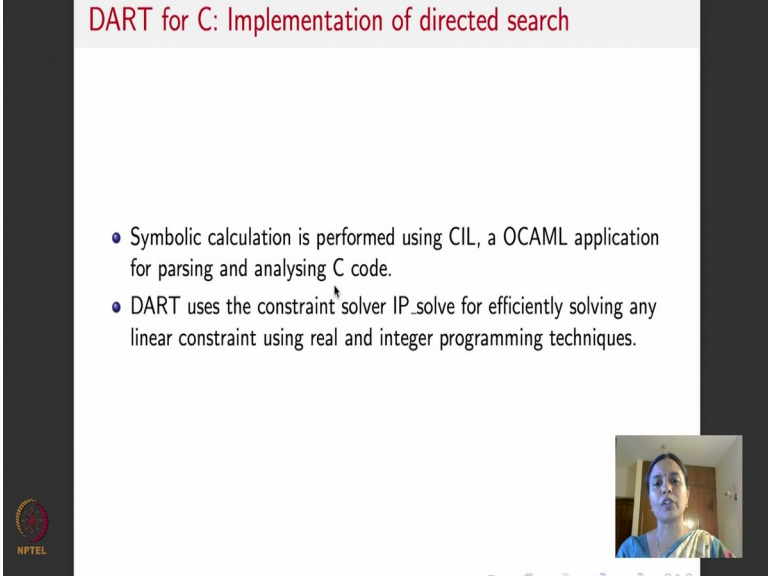
Theorem: Consider a program P (with restricted syntax). The following hold:

- 1 If run_DART prints out "Bug found" for P , then there is some input to P that leads to an abort,
- 2 If run_DART terminates without printing "Bug found", then there is no input that leads to an abort statement in P , and all paths in $Execs(P)$ have been exercised.
- 3 Otherwise, run_DART will run forever.



It can terminate saying I have found a bug which is good because it is able to reach the error statement or it can systematically explore all the program paths in a program right. So, that is what it does for C after random initialization.

(Refer Slide Time: 14:49)



The slide is titled "DART for C: Implementation of directed search" in red text at the top. It contains two bullet points: "Symbolic calculation is performed using CIL, a OCAML application for parsing and analysing C code." and "DART uses the constraint solver IP_solve for efficiently solving any linear constraint using real and integer programming techniques." In the bottom right corner, there is a small video inset showing a person speaking. The NPTEL logo is visible in the bottom left corner of the slide frame.

- Symbolic calculation is performed using CIL, a OCAML application for parsing and analysing C code.
- DART uses the constraint solver IP_solve for efficiently solving any linear constraint using real and integer programming techniques.

So, how does it do the symbolic updation of C? For C, it so happens that there is this OCAML application called CIL. So, DART uses this for doing parsing an analyzing C code. So, that it can do the symbolic update or the symbolic sate, the particular constraint solver that is reported in the DART paper is this solve called IP solve which they use, basically you could use anything any other interpreter that you will like or any other constraint solver that you like.

I will update the correct reference that contains this paper of DART which mainly works for C, and DART is actually now implemented in to a full fledged tool called CUTE as I told you in the last lecture. I will update a reference for CUTE I will also put in a reference for a related open source tool for C called CREST. CUTE is proprietary, but CREST is open source. So, that you could try out symbolic execution if you want and if you know Java if you are interested in symbolic execution for Java there is a tool called JUnit j cute sorry which runs for Java, I will upload a document that contains the references to each of these papers in the announcement section of the course so that you can look at up for further details. If you have any particular doubts specifically about the crest tool feel free to pin me because I have used it in tried at out and I might be able to help you with running and experimenting that tool to do symbolic execution for C programs.

So, I will stop here for now, we are towards the end of the course what I will be doing is I had asked you all feedback for some topics that you would want me to cover. So, in the next few lectures that are available I am going to pick a select subset of those topics especially those that I am familiar with and I will give you introductory modules on each of those.

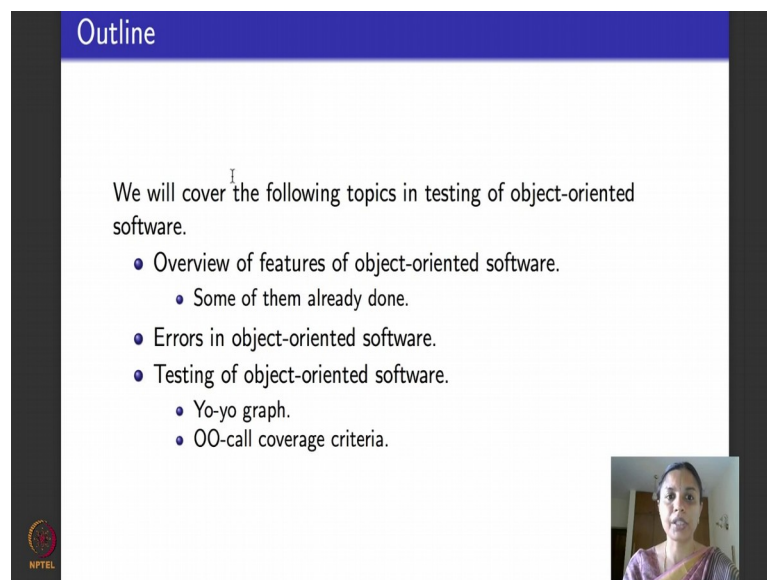
Thank you.

Software testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture - 55
Testing of Object-Oriented Applications

Hello there, welcome to the first lecture of week 12. If you remember in week 10, I had thought you about testing web applications and testing object oriented applications, system level testing of object oriented applications. This was the outline of how I began. We saw overviews of features of object oriented software, some of which were done in the middle of the course.

(Refer Slide Time: 00:24)



The slide is titled "Outline" in a blue header. The main content area is white and contains the following text and list:

We will cover the following topics in testing of object-oriented software.


- Overview of features of object-oriented software.
 - Some of them already done.
- Errors in object-oriented software.
- Testing of object-oriented software.
 - Yo-yo graph.
 - OO-call coverage criteria.

In the bottom right corner of the slide, there is a small video inset showing a woman (Prof. Meenakshi D'Souza) speaking. The NPTEL logo is visible in the bottom left corner of the slide.

Then we saw an error overview of all kinds of errors or anomalies is that can occur in a object oriented software. Especially during integration testing of object oriented systems. Then in we also saw what is a yo-yo graph. What I had not done is a part of those lectures was to do object oriented call coverage criteria specific to integration testing. In week 11, I decided to drop object oriented testing without doing this and give you an overview of symbolic and concolic test selection techniques.

Mainly because I wanted that to be a complete set of 5 lectures covering one week's details. So, what I will now do is catch up on what I did in week 10 and finish object oriented testing.

(Refer Slide Time: 01:25)




Abstraction in OO programs

OO languages use

- **Classes** to represent data abstraction.
- In addition **inheritance**, **polymorphism** and **dynamic binding** support abstraction.
- New type created by inheritance are **descendents** of the existing type.
- **Extension** and **refinement**:
 - A class **extends** its parent class if it introduces a new method name and does not override any methods in an ancestor class.
 - A class **refines** the parent class if it provides new behaviour not present in the overridden method, does not call the overridden method and its behaviour is semantically consistent with that of the overridden method.

So, to recap, we saw all the features of object oriented software: abstraction, inheritance, polymorphism, 2 kinds of inheritance, what is the polymorphic method, which is the level of object oriented testing that we are going to do, which is basically interclass and intraclass testing.

(Refer Slide Time: 01:31)



Four levels of class testing


With respect to testing that is unique to OO software, a class is usually regarded as the basic unit of testing.
There are four levels of testing classes.

- **Intra-method testing**: Tests are constructed for individual methods (traditional unit testing).
- **Inter-method testing**: Multiple methods within a class are tested in concert (traditional module testing).
- **Intra-class testing**: Tests are constructed for a single class, usually as sequences of calls to methods within the class.
- **Inter-class testing**: More than one class is tested at the same time, usually to see how they interact (kind of integration testing).

(Refer Slide Time: 01:38)

Visualizing OO interactions

- We assume that a class encapsulates state information in a collection of *state variables*. The behaviors of a class are implemented by methods that use the state variables.
- The interactions between the various classes and methods within/outside a class that occur in the presence of inheritance, polymorphism and dynamic binding are complex and difficult to *visualize*.
- We will go through some examples to illustrate the issues.



And how to difficult, the problem of difficulty in visualising object oriented interactions in the presence of class hierarchies.


(Refer Slide Time: 01:44)

Class hierarchy and method overriding: Example

```
classDiagram
    class W {
        -v
        +m()
        +n()
    }
    class V {
        -x
        +m()
    }
    class X {
        +m()
        +n()
    }
    W <|-- V
    W <|-- X
```

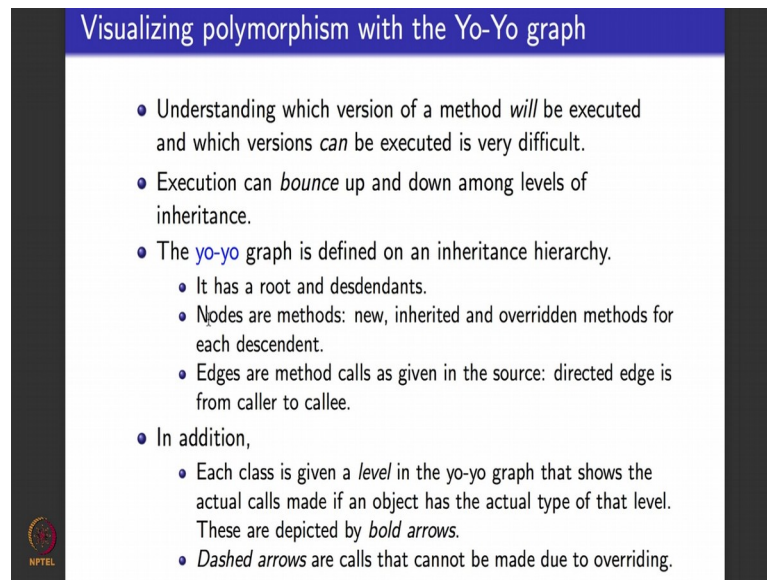
```
1. void f(boolean b)
2. {
3.     W o;
4.     ...
5.     if(b)
6.         o = new V();
7.     else
8.         o = new W();
9.     ...
10.    o.m();
11. }
```

- V and X extend W, V overrides method `m()` and X overrides methods `m()` and `n()`.
- -: attributes are private, +: attributes are non-private.
- The declared type of `o` is `W`, but at line 10, the actual type can be either `V` or `W`.
- Since `V` overrides `m()`, which version of `m()` is executed depends on the input flag to the method.



And overriding methods and polymorphic methods, and we the also saw this yo-yo graph which depicts how object oriented interactions occur.

(Refer Slide Time: 01:53)

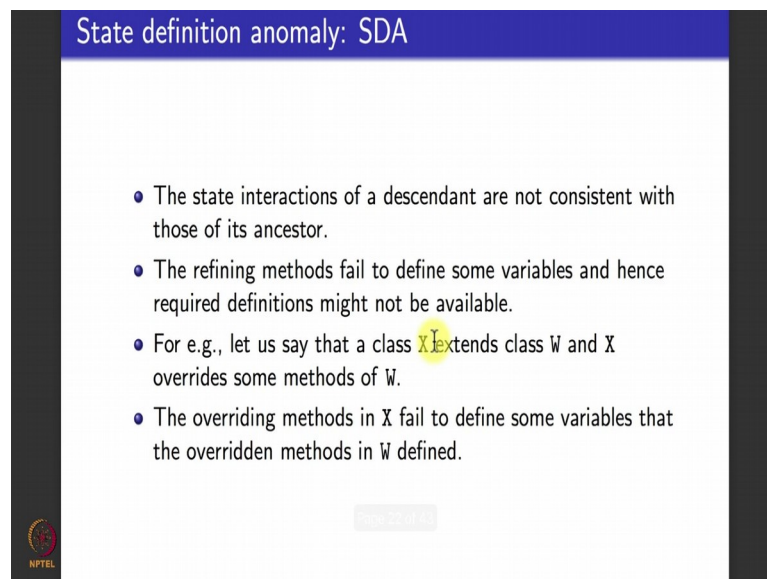


Visualizing polymorphism with the Yo-Yo graph

- Understanding which version of a method *will* be executed and which versions *can* be executed is very difficult.
- Execution can *bounce* up and down among levels of inheritance.
- The *yo-yo* graph is defined on an inheritance hierarchy.
 - It has a root and descendants.
 - Nodes are methods: new, inherited and overridden methods for each descendent.
 - Edges are method calls as given in the source: directed edge is from caller to callee.
- In addition,
 - Each class is given a *level* in the yo-yo graph that shows the actual calls made if an object has the actual type of that level. These are depicted by *bold arrows*.
 - *Dashed arrows* are calls that cannot be made due to overriding.

And in week 10, I had also introduced you to the various kinds of object oriented faults.

(Refer Slide Time: 02:04)



State definition anomaly: SDA



- The state interactions of a descendant are not consistent with those of its ancestor.
- The refining methods fail to define some variables and hence required definitions might not be available.
- For e.g., let us say that a class **X** extends class **W** and **X** overrides some methods of **W**.
- The overriding methods in **X** fail to define some variables that the overridden methods in **W** defined.

State definition anomaly, state definition inconsistency, state visibility fault, all other kinds of faults.

(Refer Slide Time: 02:18)

Coupling Sequences

- Pairs of method calls within body of method under test.
 - Made through a common *instance context*.
 - With respect to a set of state variables that are commonly referenced by both methods.
 - Consists of at least one coupling path between the two method calls with respect to a particular state variable.
- Represent potential *state space interactions* between the called methods with respect to calling method.
- Used to identify points of integration and testing requirements.



And then what I did not do is the last part, which dealt with coupling of variable specific object oriented software.



(Refer Slide Time: 02:21)

Moving on...

In the next lecture, we will look at how to test for the problematic features.

The notion of coupling will be extended to handle indirect coupling due to OO features.

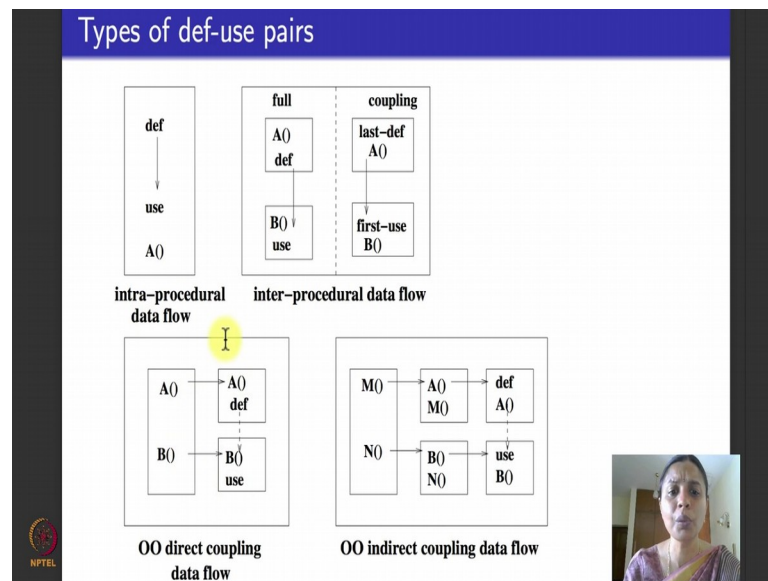
Data flow criteria will be defined for such coupling variables.



How does data flow work in the presence of coupling variables, and what are the object oriented specific coverage criteria that we would see? So, that is what I am going to do in today's lecture. Coupling variables is not a concept that is new for us in the course. We have seen coupling when we did design integration testing. Coupling variable is a variable that is used in one and defined in one module and used in the other. But now the

problem is in object oriented software, how does a coupling variable occur? There could be a pair of method calls within the body of method that is being testing. This pair of method calls can be made through a common instance of an object with respect to set a variables that are commonly referenced by both the methods. And they consist typically, of at least one coupling parts between the 2 method calls for one state variable.

(Refer Slide Time: 03:36)

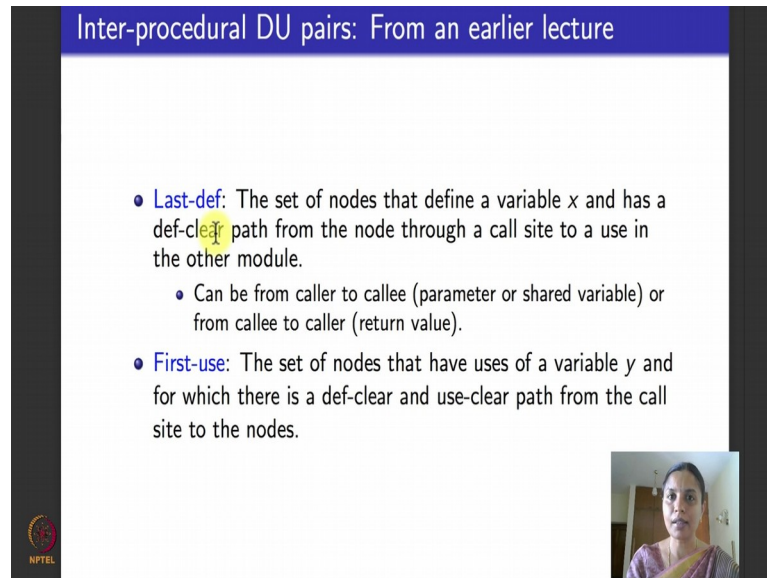


So, these represent potential state space interactions between the called methods with reference to the calling method. So, we have to identify which are the points of integration and what are the testing requirements in terms of coverage criteria for us to test on. So, we look at the various kinds of definition-use pairs that we have seen till now. This is one thing that we saw right in the beginning when we did data flow testing along with graphs. Within one method or a procedure A, a variable can be defined and used. So, this is intra procedure or intra method, within a method, normal kind of definition-use data flow. This we saw when we did data flow coverage criteria during graph base testing.

The next is inter procedural data flow which is from one procedure A to another procedure B, a particular definition happens in A and is used in B. If the definition happens to be the last definition in A and the first use of a particular variable in B, then such a variable is called a coupling variable and we consider coupling data flow. Now when it comes to object oriented data flow, what we talking about is instances of these a

and b and definitions occurring in the instance of A and in instance of B. This is direct object oriented coupling. In indirect object oriented coupling what happens? There is a method m, there is another method n, there is a method a that uses m, there is method b that uses n and the definition and use happens in a and b for an m and n. So, we see an example where this becomes little more clear.

(Refer Slide Time: 04:59)

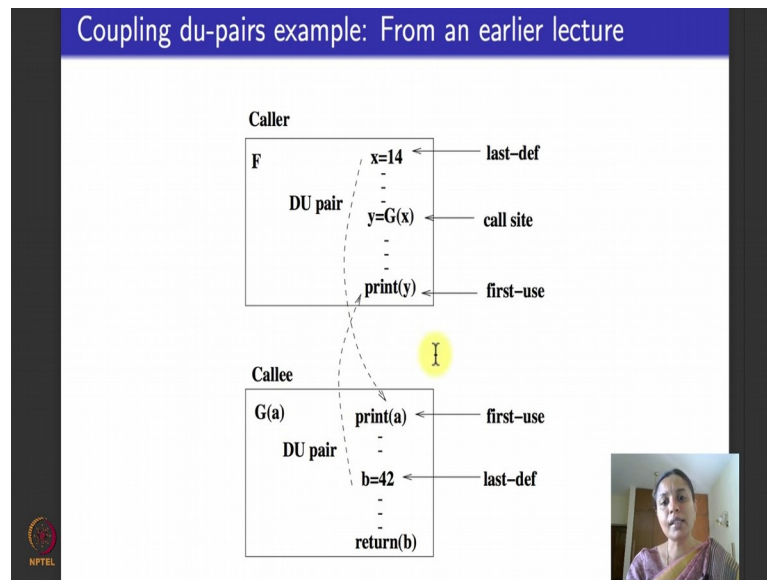


Inter-procedural DU pairs: From an earlier lecture

- **Last-def:** The set of nodes that define a variable x and has a def-clear path from the node through a call site to a use in the other module.
 - Can be from caller to callee (parameter or shared variable) or from callee to caller (return value).
- **First-use:** The set of nodes that have uses of a variable y and for which there is a def-clear and use-clear path from the call site to the nodes.

So, this is again a recap from the earlier lecture. What is the last def for a last definition? It is a set of nodes that define a variable x and has a def clear path from the note through a call site to a use in other procedure or module. That can be from the caller to a callee or can be from callee back to the caller, in which case it is the value that is returned. What is the first use? It is a set of nodes or statements that have uses of a particular variable y for which there is a def clear and a use clear path from the call site to the node that contains this use.

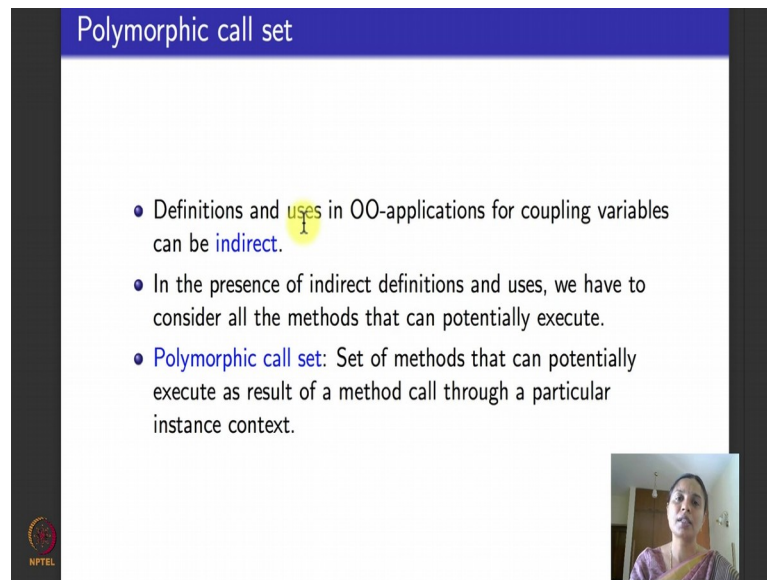
(Refer Slide Time: 05:42)



This is what we saw a long ago when we did design integration testing. And this if you remember is a exact recap of a example that I used in the earlier lecture. There is a procedure F that it some point in it is code calls another procedure G, and it calls G with x the value that G returns is passed to y. So, this statement is what is called the call site.

This the last definition of x that occurred just before this call is the last definition, when it is passed it becomes the first use and sum calculation happens and the value that is a returned, the statement that this causes the value that to be returned, is what is called the last definition.

(Refer Slide Time: 06:30)



The slide is titled "Polymorphic call set" in a blue header. It contains three bullet points:

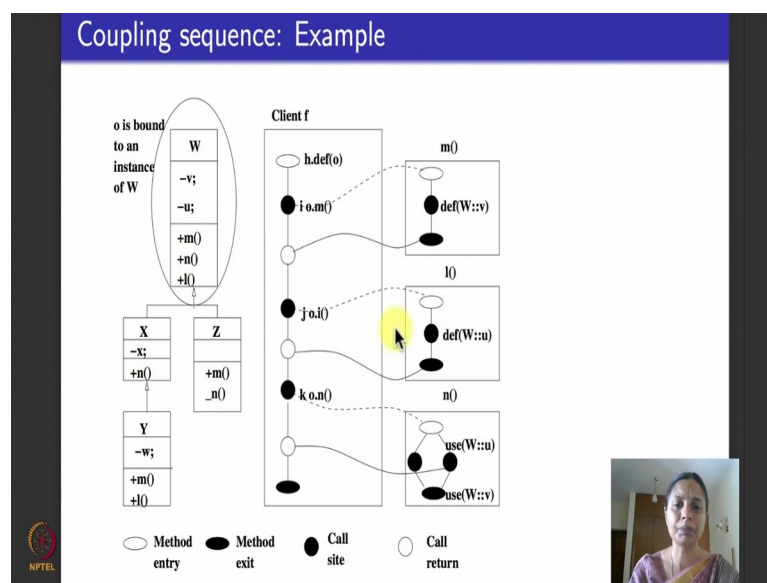
- Definitions and uses in OO-applications for coupling variables can be indirect.
- In the presence of indirect definitions and uses, we have to consider all the methods that can potentially execute.
- **Polymorphic call set:** Set of methods that can potentially execute as result of a method call through a particular instance context.

In the bottom right corner, there is a small video inset showing a woman speaking. The NPTEL logo is visible in the bottom left corner of the slide.

So, this is what we saw as last definition and first use. Now we will see the same concept, but with reference to object oriented testing. So, to understand that we first need to understand when we have polymorphic methods, what is a polymorphic call set? So, here in the presence of the polymorphic methods, the first thing to observe is that the definitions and use for such coupling variables like these can be indirect. So, what do we mean by indirect? Indirect means you do not know which version will be executed in which method that is being used. So, in the presence of indirect definitions and uses, we have to consider all the methods that can potentially execute.

So, polymorphic call set is a set of methods that can potentially execute as a result of a method call through a particular instance context.

(Refer Slide Time: 07:11)

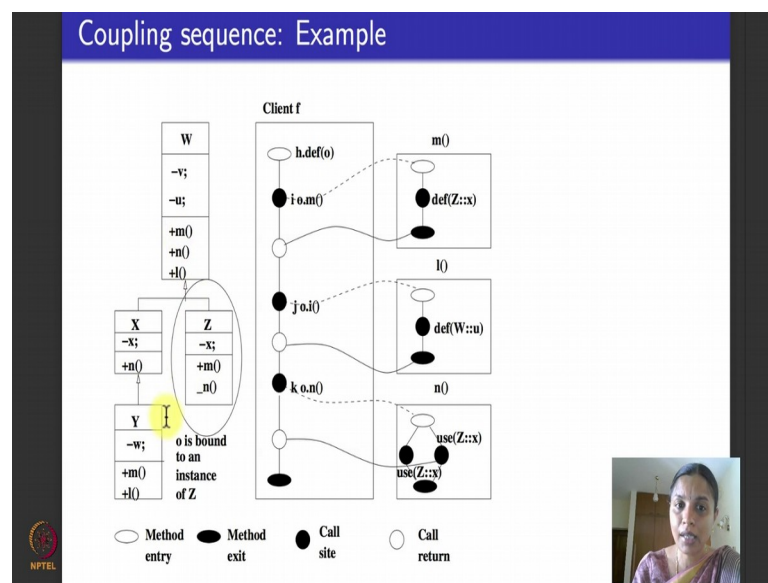


I will illustrate with the example. So, this is a slightly big figure, but we will go through it slowly. On the left hand side is what we have the class hierarchy. There are 4 classes w, x, z and y. w has 2 private variables v and u, 3 methods m n and l. x has a variable small x method n again over ridden. z has no variables that matters to us for now. So, we will ignore the other variables that do not matter to us in z, I have not depicted it. And z also has 2 methods m and n, and y has a local private variable w and 2 methods m and l. Please remember that m occurs here, here and here. And n occurs in w, x and z, and l occurs in w and y. So, here is a typical code that uses this class hierarchy and the methods from these various classes.

Let us say it is the code of some client f. So, there is an object o. Let us say now that o is bound to an instance of the class w for our example in this code. So, what happens now let see at this point the method m is called in o and m is executed if you go to m m let say defines a v now remember object is bound to an instance of w. So, it is w's definition of v that is used and passed back. And then moving on, so how do I read this figure, what is the legend? These white eclipses are method entries, these black eclipses are method exits. So, there are method entries here, here and here for m, l and n. There is a main method entry here and method exit here. These black circle ones are call sites where the method calls happens and the white circles are when the method finishes executing, it is the call return.

So, from this figure I hope it will be clear. So, method m is call which defines v and the value is returned and the method i is called which defines u and then the values returned and then the method n is called which defines which uses u, and then let us say it also uses v and it is returned. So, this dotted, dashed lines and the solid lines represent the coupling sequence of variables and the call sites and call returns. This happens for this class hierarchy when the object o is bound to the class w.

(Refer Slide Time: 09:49)





Let us say an object o is instead for the same class hierarchy, is bound to an instance of the class z. Then what happens? Right up front when this client call this method i, instead of taking w's definition let us say z's definition of x is taken, we are introduced to variable x that matters to us in z. And then let us say the next call w's definition of u is taken because n, l does not belong to z. And then x's use in z is considered and x's use in z is again considered here right. So, in which case the problem is because z does not have an instance of l w's l is considered and it could cause the coupling sequence being different. There is branching here which did not happen here during the use. When we come to object oriented coupling, what are a testing goals? We want to be able to test the following.

(Refer Slide Time: 10:40)

OO coupling: Testing goals

- We want to test how a method can interact with instance bound to object o .
 - Interactions occur through the coupling sequences.
- Need to consider the set of interactions that can occur.
 - What types can be bound to o ?
 - Which methods can actually execute? (polymorphic call sets).
- Test all couplings with all type bindings possible.





We want to test how a method can interact with an instance bound to an object o . Like for an example here an object o is bound to an instance of w , the interaction is different from when an object o is bound to an instance of z . And we need to consider the set of interactions that can occur, what are the types that can be bound to o , which are the methods that can actually executed the presence of polymorphic call sets, and we have to test all couplings with all the type bindings.

(Refer Slide Time: 11:12)

All-Coupling Sequences

- **All-Coupling-Sequences (ACS)**: For every coupling sequence S_j in $f()$, there is at least one test case t such that there is a coupling path induced by $S_{j,k}$ that is a sub-path of the execution trace of $f(t)$.
- At least one coupling path must be executed.
- This does not consider inheritance and polymorphism.





So, we define 4 coverage criteria. The first one that we will look at is what is called all coupling sequences abbreviated as ACS, where for every coupling sequence s_j in the particular method f , there is at least one test case t is needs to be written such that there is a coupling path introduced by s_j , induced by $s_j k$ which is a sub path of the execution trace of f when it is executed in t . So, what it said is the there is a coupling sequences s_j for example, like this kind of sequence, what is coupled with what, what is coupled with what, what is coupled with what. Every dash line is coupled with the following bold normal line. So, for every sub sequence in f , there is at least one test case that we must write such that when you execute the test case the coupling parts comes $s_j k$ that should be a sub path of the execution trace when f is executed with t . Which means what? At least one coupling path must be executed. Let us say there are 3 coupling variables, all it says is that per variable please execute at least one coupling path. No inheritance, polymorphism, polymorphic call set, it is not considered in this coverage criteria.

(Refer Slide Time: 12:26)

All Poly Classes

- **All-Poly-Classes (APC):** For every coupling sequence $S_{j,k}$ in method $f()$, and for every class in the family of types defined by the context of $S_{j,k}$, there is at least one test case t such that when $f()$ is executed using t , there is a path p in the set of coupling paths of $S_{j,k}$ that is a sub-path of the execution trace of $f(t)$.
- Includes instance contexts of calls.
- At least one test for every type the object can bind to.
- Test with every possible type substitution.





The next is all polyclasses abbreviated as APC. Here for every coupling sequence $s_j k$, in the method that is the same as we saw here. And for every class in the family of types defined by the context of $s_j k$, there is at least one test case t such that when is executed on f there is a path in the set of coupling paths that is the sub path of the execution trace of f of t . So, what is the difference between this and this? This includes all the coupling variables, this includes all the coupling variables in the context of the call that was made considering the binding also. So, it includes at least one test case for every type the



objective can be bound to. If you go back to the example, it will include one test case for this case when o is bound to z , one test case for this case when o is bound to w . And it also test with every possible type substitution that can happen. Obviously, this is more effective or it subsumes the all coupling sequences criteria.

So, the next is all coupling defs and uses. So, here what happens? Abbreviated as ACDU, for every coupling variable v in each coupling pair $s\ j\ k$ of a test case t , there is a coupling path introduced by $s\ j\ k$ such that p is the sub path of execution trace of f of t for at least one case t . Which means what? Every last definition of a coupling variable reaches every first use. So, here it was like at least the first case, here it is every. This is again without inheritance and polymorphism. So obviously, the next one is going to be every last definition reaching every first use with inheritance and polymorphism.

(Refer Slide Time: 14:11)

All poly coupling defs and uses

- **All-Poly-Coupling-Defs-and-Uses (APDU):** For every coupling sequence $S_{j,k}$ in $f()$, for every class in the family of types defined by the context of $S_{j,k}$, for every coupling variable v of $S_{j,k}$, for every node m that has a last definition of v and every node n that has a first-use of v , there is at least one test case t such that when $f()$ is executed using t , there is a path p in the coupling paths of $S_{j,k}$ that is a sub-path of the trace of $f()$.
- Every last definition of a coupling variable reaches every first use for every type binding.
- Combines previous criteria.
- Handles inheritance and polymorphism.
- Takes definitions and uses of variables into account.

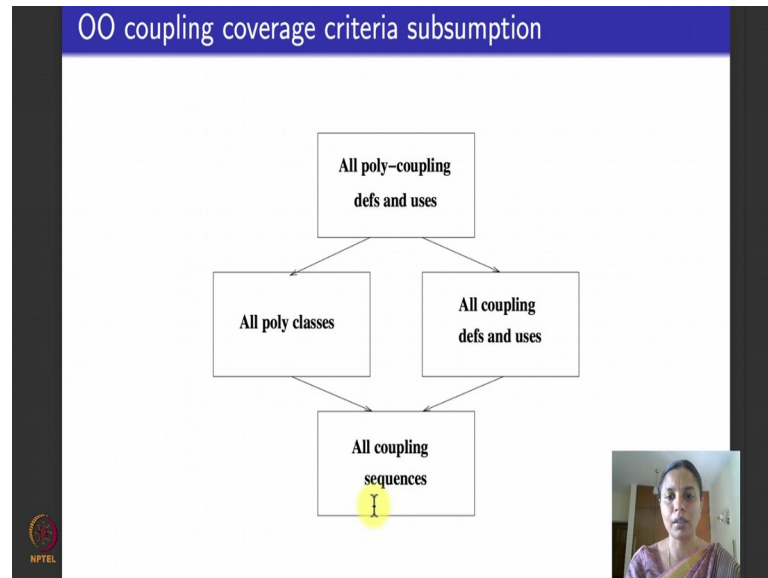



And this is the mother of all criteria. It includes polymorphism, polymorphic call sets and it says every coupling variable, every definition to every use has to be covered.

Do not worry about this definition being long. What it basically says, is it says t if I have a coupling sequence in a function f , then for every class in the family of types define by the context of $s\ j\ k$, consider every coupling variable in that and then consider the nodes in which the last definition happen and the first use happen. You must have a test case for including that path, that is what it says. So, the simplest, to repeat, is all coupling sequences which says at least one coupling path must be executed. The next is at least

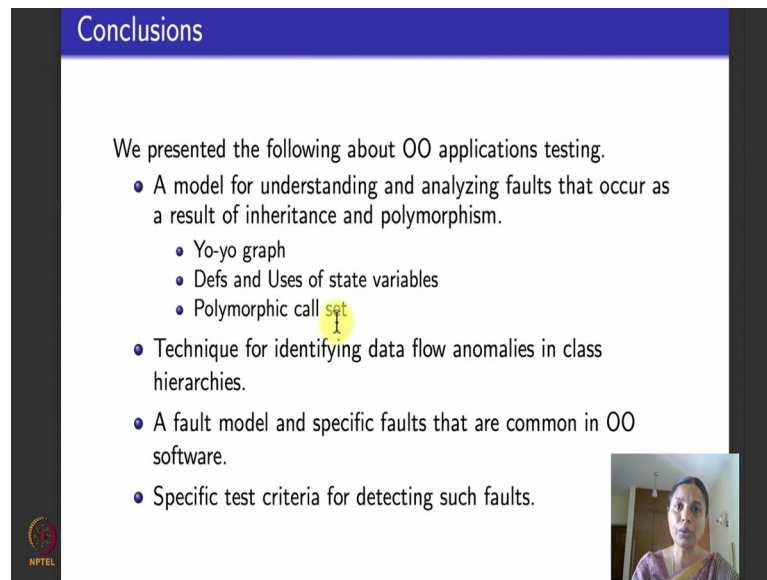
one coupling path in the presence of instance contexts and object binding should be executed. This is every last definition to every first use, reachability should be tested, without inheritance and polymorphism.

(Refer Slide Time: 15:35)



This is every last definition to every first use with inheritance and polymorphism and type binding. So, this is the highest coverage criteria, the most difficult to subsume, and by definitions, this picture should be very clear. This says that test all coupling sequences the first one. This says with polymorphism. This says all definitions and uses, but without polymorphism. This is all definitions uses with polymorphism, with object binding. So, these are the 4 coupling coverage criteria that you can use for integration testing for object oriented software.

(Refer Slide Time: 16:02)



The slide is titled "Conclusions" in a blue header. The main content area is white and contains the following text and list:

We presented the following about OO applications testing.

- A model for understanding and analyzing faults that occur as a result of inheritance and polymorphism.
 - Yo-yo graph
 - Defs and Uses of state variables
 - Polymorphic call set
- Technique for identifying data flow anomalies in class hierarchies.
- A fault model and specific faults that are common in OO software.
- Specific test criteria for detecting such faults.

In the bottom right corner, there is a small inset video showing a woman speaking. In the bottom left corner, there is a small NPTEL logo.

So, to conclude the object oriented software lecture we understood in week 10, yo-yo graphs. In, sorry, errors and anomalies in object oriented testing. Today we understood polymorphic call sets, what are it is definitions and uses of state variables for integration testing and also looked at various coverage criteria that will detect faults that come because of object oriented features like inheritance and polymorphism.

So, I hope this lecture is beneficial to you. This will bring us to an end of object oriented integration testing.

Thank you.

Software testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 56
Testing of Mobile Applications

Hello there we are in week 12. Towards the end of the course kept week 12 open and I had asked suggestion from course registrations of what you would want to know about testing, and a few of you replied back in the form saying, please give us an overview of mobile applications testing. So, this lecture is oriented towards that, you I will give you an overview of mobile apps testing. Like all other testing fields this is heavily domain dependent, tool dependent, and given in that we are in the last week of the course and the focus of the course has been on algorithms, I will restrict myself to giving you a reasonably thorough over view, but we will not go delve into the details. So, this is a lecture on testing of mobile applications.

(Refer Slide Time: 01:00)




So, here is what we are going to do in this lecture, I will introduce you to mobile applications testing. We will understand what are the key challenges that is being addressed when it comes to testing of mobile apps and what are the various types of mobile apps testing.

Mobile apps testing is based on entities called emulators and tools. So, I list lend the

course with giving you an introduction of some of the good emulators that are open source, that I know about feel free to try them out if you have your mobile app to test.

(Refer Slide Time: 01:35)



Mobile apps testing

- Mobile application testing is a process by which application software developed for handheld mobile devices is tested for its functionality, usability and consistency.
- Can be automated or manual.
- Mobile applications either come pre-installed or can be installed from mobile software distribution platforms.
- Related area: Testing of wearable applications.

NPTEL

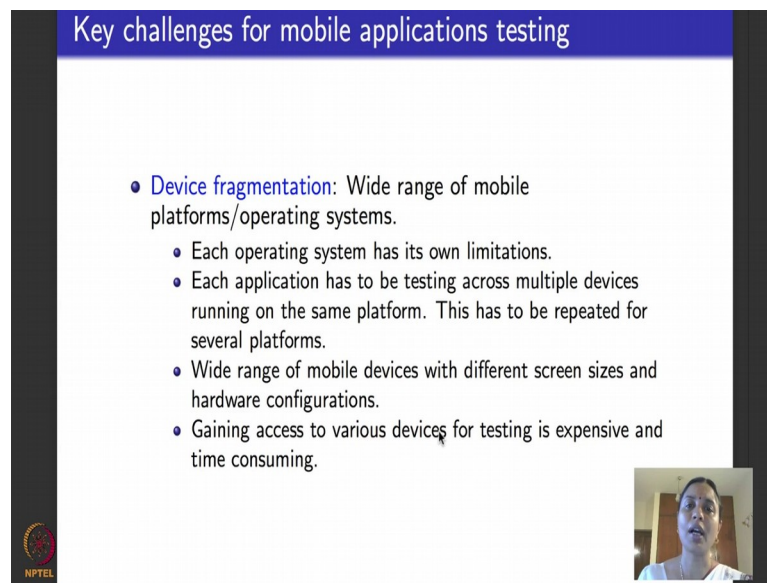
So, we begin with an over view of mobile apps testing, the challenges in mobile apps testing. So, what is a mobile application testing? So, like any other piece of software it is yet another software program. It is written in a particular programming language, meant to run a particular operating system, takes user inputs, produces outputs, it is like any other piece of software. But it is a mobile application, which means it is the software that is meant to run on these hand-held devices on a particular platform given by the mobile device, tuned to a particular operating system which is obviously, not as big as a large server or a desktop application.

So, mobile application testing is a process by which an application software like an app that is developed for hand-held devices is tested, for all the features. It should do what it is supposed to do, which is basically functionality and because it is mobile app it is usability factor must be very high, because it is important for the success of the app and it should be consistent. Consistent in the sense that it should be consistent across a Android platform, versus a windows platform versus a IOS platform also. Mobile application testing, like all other testing, can be automated can be manual also. Mobile applications, they are either pre-installed when you buy a particular mobile phone or device or they can be downloaded and installed from any of the mobile software

distribution platforms.

One of the biggest platforms that we are aware of is Google play. A related area of testing is testing of variable applications which are this fit bits, Google, apple watch Google glasses. All these wearable devices, which also have elementary computing power. Please note that we are not looking at testing of wearable applications, but some of the things that we will see today is also relevant to testing of wearable applications.

(Refer Slide Time: 03:36)



The slide is titled "Key challenges for mobile applications testing" in a blue header. It contains a bulleted list of challenges. In the bottom right corner, there is a small video inset showing a man speaking. The NPTEL logo is in the bottom left corner.

- **Device fragmentation:** Wide range of mobile platforms/operating systems.
 - Each operating system has its own limitations.
 - Each application has to be testing across multiple devices running on the same platform. This has to be repeated for several platforms.
 - Wide range of mobile devices with different screen sizes and hardware configurations.
 - Gaining access to various devices for testing is expensive and time consuming.

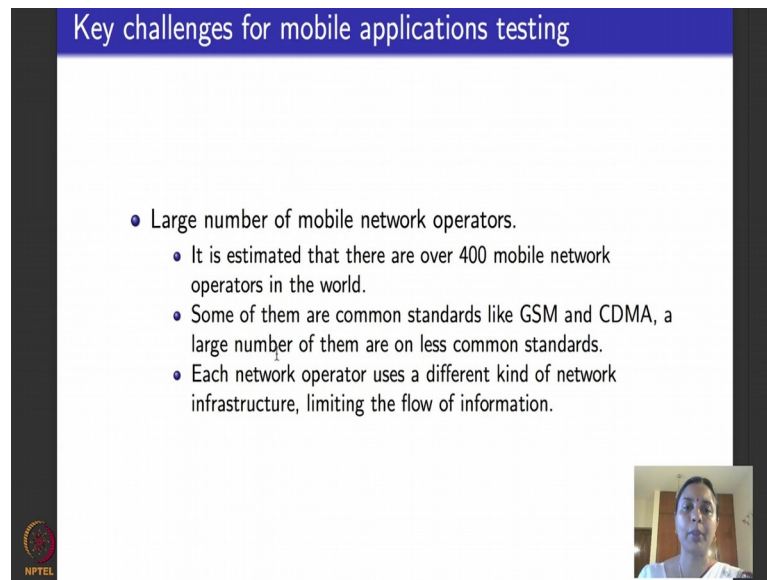
So, we will begin with what are the challenges when it comes to mobile applications testing. So, as I told you a mobile application is another piece of software written in a particular programming language. So, it goes through all the stages of development; unit testing, software integration testing. For unit testing software integration testing, you can do all that we did in the past. You could model it as a graph, you could use logical predicates, you could use mutation any of the things that we learned can be used.

When it comes to now after integration testing, we are talking about system level testing which is an app for working on a phone, after being downloaded and installed and being made available on a phone, which is system level testing for a mobile application. That is what we are going to focus on. So, when it comes to the system level properties or system level testing of a mobile application, what are the key challenges that are there? The biggest problem is what is called is device fragmentation problem in this area. So, what is it mean? It basically means that there are so many mobile phones right, so many

different mobile phones, very very expensive phones very very cheap phones. Phones that are feature rich, phone that have a good camera, phones that are very cheap and so on. Huge range of operating systems, platforms and phones. So, the problem is each operating system has its own limitations and each application that I develop, right. If I have to develop a particular app it could be a gaming app, it could be something, it has to work across platforms. There are of course, apps that are specifically tune for working on an IOS or an Android. But most of the common apps that we develop we expected to work across platforms, across operating systems and it has to be tested across multiple devices running on the same platform. If I take android which is by and large the most common mobile platform mobile OS, there may be a 100 different kinds of devices, that use android each with its own limitation and capability. So, a particular app needs to be tested across each of these devices, the other big problem is when you say you have to test particular app across each of these devices, a difficult to implement strategy would be to go and buy that device.

Every day there is a new phone coming in the market, how are you going to go on buying devices? If you have a particular app tested across each kind of device even across manufacturers, there are so many different phone manufacturers, it is very difficult to actually buy the phone and test it. So, that is why we resort to these things called emulators, which we will see later in this lecture. So, each of these mobile devices what are their problems? They have different screen sizes, different resolutions, they have different memory, different hardware capability, different battery capability, everything is different in each of them.

(Refer Slide Time: 06:36)



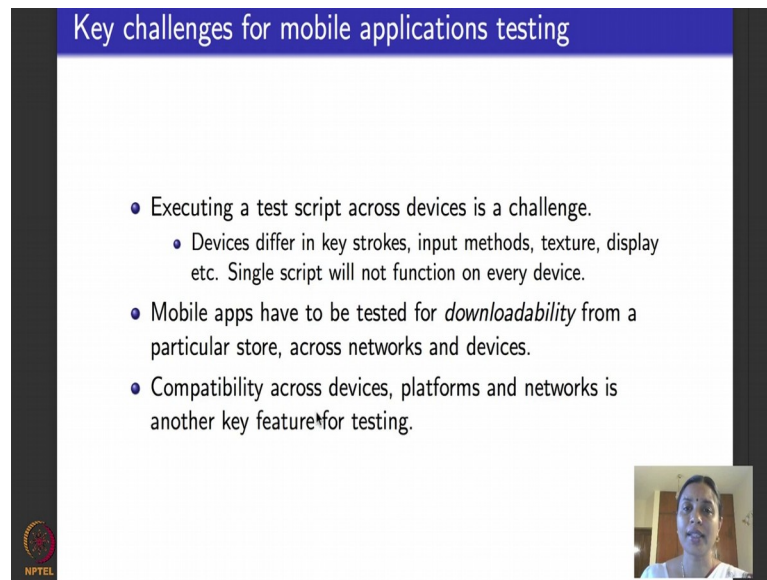
The slide is titled "Key challenges for mobile applications testing" in a blue header. It contains a bulleted list of challenges. In the bottom right corner, there is a small video inset showing a person speaking. The NPTEL logo is in the bottom left corner.

- Large number of mobile network operators.
 - It is estimated that there are over 400 mobile network operators in the world.
 - Some of them are common standards like GSM and CDMA, a large number of them are on less common standards.
 - Each network operator uses a different kind of network infrastructure, limiting the flow of information.

So, the next big challenge for mobile application testing is the very large number of network operators. It is believed that there are about 400 mobile network operators across the world. It looks like a small number, but from the point of view of testing, it is a large number. Some of them use common standards that you might be aware of like GSM, CDMA and all that, but if you see a good number of them are also on less common standards.

So, when somebody operates a mobile network, it could be an Airtel or it could be a Jio or whatever it is if that company is using a common standard, then you say I test the mobile network as per that standard. But a good number of the network operators use less common standards. So, across the networks that the operating you have able to be able to test your app, they use a different kind of network infrastructure. So, there is a challenge about how information flows from one network to the other, all these things matter a lot.

(Refer Slide Time: 07:33)



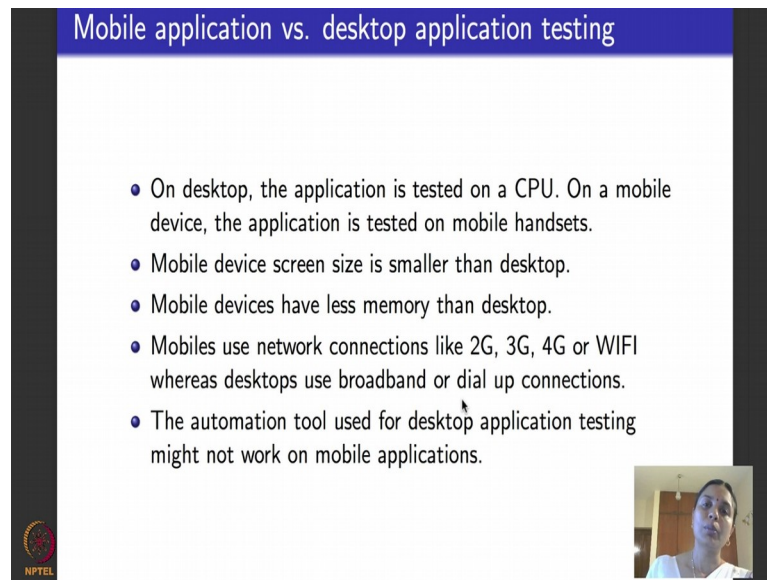
The slide is titled "Key challenges for mobile applications testing" in a blue header. It contains a bulleted list of three main challenges. The first challenge is "Executing a test script across devices is a challenge," which includes a sub-bullet: "Devices differ in key strokes, input methods, texture, display etc. Single script will not function on every device." The second challenge is "Mobile apps have to be tested for *downloadability* from a particular store, across networks and devices." The third challenge is "Compatibility across devices, platforms and networks is another key feature for testing." In the bottom right corner of the slide, there is a small video inset showing a person speaking. The NPTEL logo is visible in the bottom left corner of the slide frame.

- Executing a test script across devices is a challenge.
 - Devices differ in key strokes, input methods, texture, display etc. Single script will not function on every device.
- Mobile apps have to be tested for *downloadability* from a particular store, across networks and devices.
- Compatibility across devices, platforms and networks is another key feature for testing.

So, the next set of challenges are, as I told you, executing a, actually executing; suppose you have a test case that is developed and packaged as a test script, then you ready to test. So, you need to be able to execute it on a particular device. In a desktop it is fairly common, if you have a Windows desktop or a Ubuntu desktop, you can go ahead and write scripts configure tool like JUnit for example, to write test scripts, but that is not the same when you are executing a test script on a mobile phone.

Why is it not the same? Because each different kind of phone or a device differ in several things, they differ in their keystrokes, their input methods, their texture, their display, you cannot write a single script that will ensure execution across all phones even if the platform or the OS is common, it is very difficult. The other thing that is very specific to mobile apps, like web apps also, is that, they have to be tested for what is called and download-ability. From a particular store, if they are available let us say from Google play store, then it has to be down-loadable across networks across devices. So, that is a challenge that is very specific to mobile phones. The other thing that people test is what is called is compatibility which we discussed here right in this part. You have to be able to ensure that a particular app if you claim it is compatible across devices platforms and networks. So, these are the challenges.

(Refer Slide Time: 09:04).

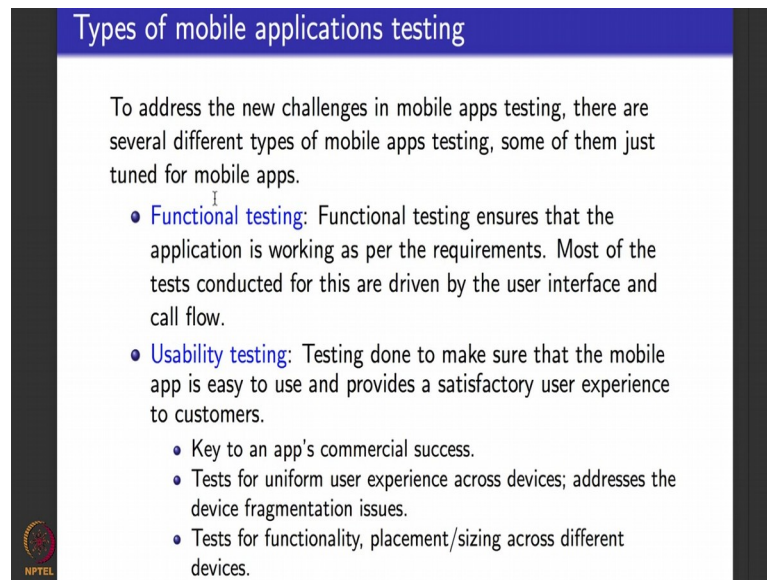


The slide is titled "Mobile application vs. desktop application testing" in a blue header. It contains a bulleted list of five points comparing mobile and desktop testing environments. In the bottom right corner, there is a small video inset showing a person speaking. The NPTEL logo is visible in the bottom left corner of the slide frame.

- On desktop, the application is tested on a CPU. On a mobile device, the application is tested on mobile handsets.
- Mobile device screen size is smaller than desktop.
- Mobile devices have less memory than desktop.
- Mobiles use network connections like 2G, 3G, 4G or WIFI whereas desktops use broadband or dial up connections.
- The automation tool used for desktop application testing might not work on mobile applications.

So, before we move on and look at how typically, what are the typical means system level features that mobile apps are tested for, let us look at what are the biggest differences between the mobile application testing and desktop application testing. On a desktop, the application is tested by using a CPU. We all know that. But on the mobile phone the application is tested on mobile handsets. Mobile device screen is usually much smaller than a desktop screen. So, as I told you test script execution across devices is a challenge. Similarly mobile devices have lot less memory than a typical desktop, that is also a challenge, apps have to be developed for working within that memory. So, memory testing is a very important feature, we will look at that also. Mobile uses, I told you different connections, 2G, 3G, 4G Wi-Fi whatever you want, desktop they use broadband or dial up or maybe Wi-Fi, they do not use anything else. Automation tool for desktop application testing will not work on mobile application, because the test script that is written for desktop is not easy to replicate to work on different kinds of mobile devices.

(Refer Slide Time: 10:18)



The slide is titled "Types of mobile applications testing" in a blue header. The main text states: "To address the new challenges in mobile apps testing, there are several different types of mobile apps testing, some of them just tuned for mobile apps." Below this, there are two bullet points: "Functional testing" and "Usability testing". The "Functional testing" bullet point describes it as ensuring the application works as per requirements, driven by the user interface and call flow. The "Usability testing" bullet point describes it as testing to ensure the app is easy to use and provides a satisfactory user experience. It includes three sub-bullets: "Key to an app's commercial success", "Tests for uniform user experience across devices; addresses the device fragmentation issues", and "Tests for functionality, placement/sizing across different devices". An NPTEL logo is visible in the bottom left corner of the slide.

Types of mobile applications testing

To address the new challenges in mobile apps testing, there are several different types of mobile apps testing, some of them just tuned for mobile apps.

- **Functional testing:** Functional testing ensures that the application is working as per the requirements. Most of the tests conducted for this are driven by the user interface and call flow.
- **Usability testing:** Testing done to make sure that the mobile app is easy to use and provides a satisfactory user experience to customers.
 - Key to an app's commercial success.
 - Tests for uniform user experience across devices; addresses the device fragmentation issues.
 - Tests for functionality, placement/sizing across different devices.

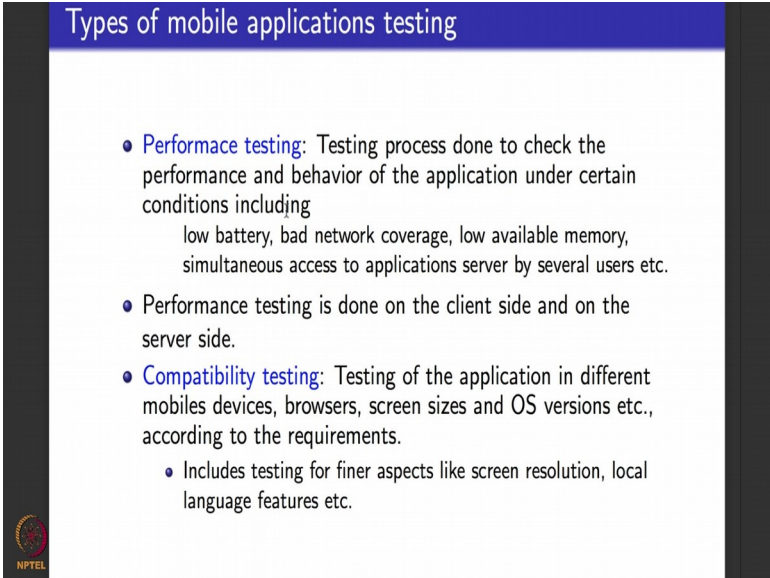
So, what are the main types of mobile application testing? So, when I go and talk to you about types of mobile applications testing, please keep one thing in mind, a mobile app is a piece of software as I told you a little while ago. So, it has to be written in a programming language, unit tested, integration tested and let us say you have it ready. Now I am considering system level testing, which is I am taking the app as a whole, packaged, ready to be put in a mobile device and tested for various aspects. So, that is the stage I am and when I discuss types of mobile application testing, that is what I am talking to you about. So, the usual thing we always do for every kind of software which is functional testing, we also do for mobile application.

What is functional testing do, it ensures that the application is working as per the specified requirements right. It may provide this main functionality; let us say you have an app that talks about the traffic, that it does tell you accurate traffic correctly. Most of the test conducted for this are driven by user interface and call flow. As I told you can use any standard testing tool to be able to functional testing. One another important system level testing for mobile app is that of usability testing. What does the word usability tell you? It is one of the quality attributes this is make done this is testing that is done to make sure that the mobile app is easy to use, and it provides a satisfactory user experience to customers. Most of us react that way right we say I do not like this app because it is UI is not good. UI is user interface, we say I do not like it is difficult to use, it is not pleasing, I don't like it and there is also a problem of native language UI right,

the same app that works here let us say a bank that operates across several different countries.

Let us say you know it mobile app for that bank which provide similar features has to be in English for a particular country, and has to be in the national language for another country, let us say it could be French or German or not it can even be in Hindi right. So, the same app has to be usable. Usability is it is to be easy to use it should be usable at all, and not really easy to use user was using it should be satisfied with the UI. So, this is very important because it is one of the key factors to an app's commercial success, and it should test for uniform user experience across devices. You cannot claim and say that I haven app that looked very good only on iOS, it does, its UI does not look very good on android, that kind of apps do not sell. It also, usability testing also tests for functionality. Because it is functionality of a mobile app is typical though interaction, it tests for placement and sizing that of the various things across devices.

(Refer Slide Time: 13:14)



The slide is titled "Types of mobile applications testing" in a blue header. It contains two main bullet points, each with a sub-bullet. The first bullet point is "Performance testing" and the second is "Compatibility testing". The NPTEL logo is in the bottom left corner.

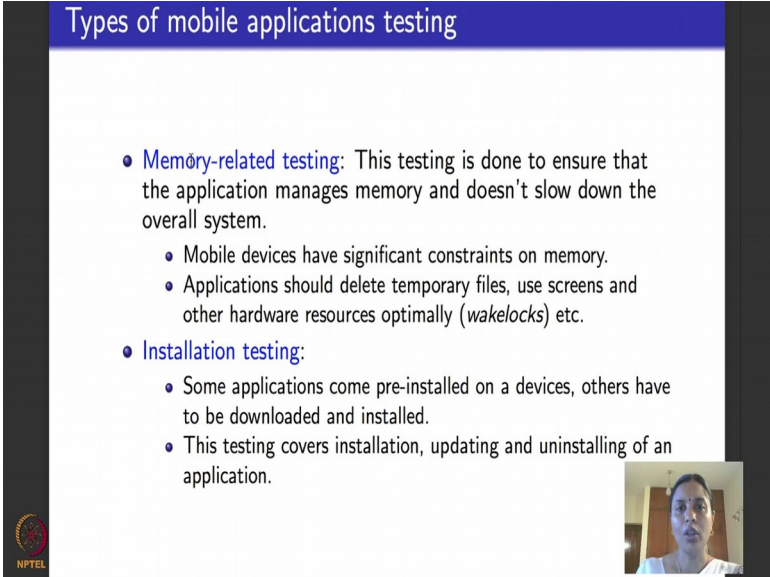
- **Performance testing:** Testing process done to check the performance and behavior of the application under certain conditions including
 - low battery, bad network coverage, low available memory, simultaneous access to applications server by several users etc.
- Performance testing is done on the client side and on the server side.
- **Compatibility testing:** Testing of the application in different mobiles devices, browsers, screen sizes and OS versions etc., according to the requirements.
 - Includes testing for finer aspects like screen resolution, local language features etc.

The next kind of important system level mobile app testing is performance testing. Here it does as always, testing, performance testing is testing done to check the performance and behavior of the application under certain conditions. What are conditions specific to mobile phones? One common conditions specific to mobile phones that we all would have encountered is low battery. At the drop of the hand mobile phones run out of battery. The next common problem that we all face is bad network; poor network

coverage; low memory available because there are several other applications running in parallel right, so that several others were simultaneously accessing the server. So, these are places that can slow down the performance of a mobile app.

So, performance testing for mobile applications focuses on these specific issues. It has been done both on the client side, which is a particular phone and on the server side which is the server where the application resides on it keeps and fetches the data. Like for example, of Facebook mobile app needs to talk to the Facebook server, Whatsapp mobile app needs to talk to the Whatsapp server, a bank mobile app need to talk to the bank server. The client is the phone the application on the phone the server is the database or the main server that application is talking to fetch data. Performance testing is dependent on both the client and the server and is tested on both sides. As I told you compatibility, Compatibility is, will in a mobile app work in different mobile devices across browsers, for several different screen sizes different mobile OS s. Will it have the correct resolution as I told you, will it have local language features, and so on, several different aspects have to be considered.

(Refer Slide Time: 15:05)



The slide is titled "Types of mobile applications testing" in a blue header. It contains two main bullet points, each with sub-bullets. The first bullet point is "Memory-related testing" and the second is "Installation testing". In the bottom right corner, there is a small video inset showing a person speaking. The NPTEL logo is in the bottom left corner.

- **Memory-related testing:** This testing is done to ensure that the application manages memory and doesn't slow down the overall system.
 - Mobile devices have significant constraints on memory.
 - Applications should delete temporary files, use screens and other hardware resources optimally (*wakelocks*) etc.
- **Installation testing:**
 - Some applications come pre-installed on a devices, others have to be downloaded and installed.
 - This testing covers installation, updating and uninstalling of an application.

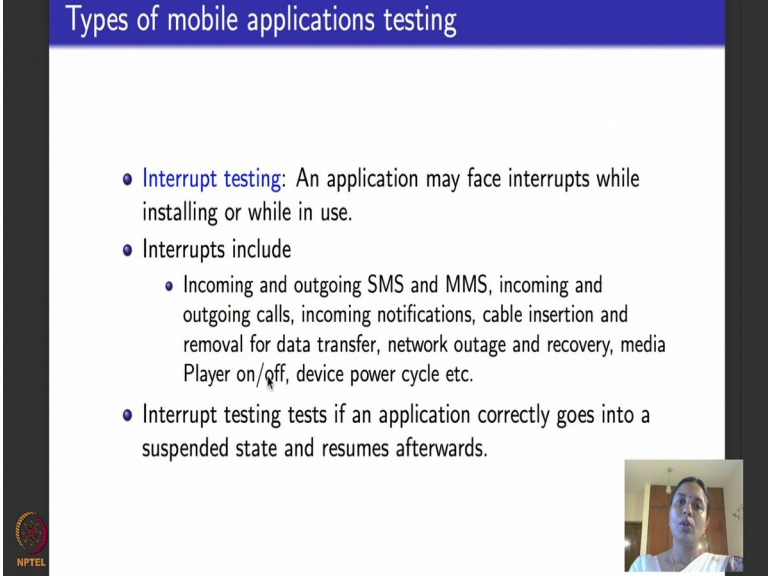
Moving on, what are the other specific system levels testing corresponding to mobile apps? The other one I told you is this is one of the problems that we all face: runs out of battery very slow, slows down. So, that is what is called memory related testing. We know that mobile devices have significant constraints on memory, because that is the

best memory that is being provided at a particular cost is already been provided by the manufacturer. So, mobile related testing is testing done to ensure that the particular app manages memory, and does not slow down the overall system. What is it mean to manage memory? For a mobile app, when it is, say managing memory, it should mean that they have to delete temporary files, uses screens and other hardware resources optimally. What do I mean by optimally? So, if u developed a mobile app, you would have you would be aware of a feature called a wake locks.

Basically wake lock tells you what to keep active and what to suspend. So, for example, let us say you have Whatsapp running, and Whatsapp does not have to keep this screen on the screen brightness and it on all the time, for you to for it to be able to notify when a new message is coming. It will have a small thread running somewhere in the background inside your phone which will look for messages in your inbox, and when the messages come it will pop it. Whereas, let us say you have another application which is playing a movie right on your phone, then for that you need to keep the screen on. So, wake locks are things that you tried as a part of your mobile app, which tell you what to keep on in terms of the hardware resources, screen is on or off, volume is on or off, display is on or off brightness is so much and so on. So, managing wake locks is an important part of memory usage.

If you keep lot of things on that you do not need all the time then the battery phone and memory is going to drain and that is not good for the app's performance. So, more memory related testing tests for all these optimal usage of resources. The next is installation testing. As I told you when it comes to mobile apps, with some of them come pre-installed on a particular device, where as others to be download and install the particular device. So, this kind of testing covers installation updating of an app which we do very frequently periodically, both system level functionality and a particular related app and un-installing also is typically covered in installation testing.

(Refer Slide Time: 17:45)



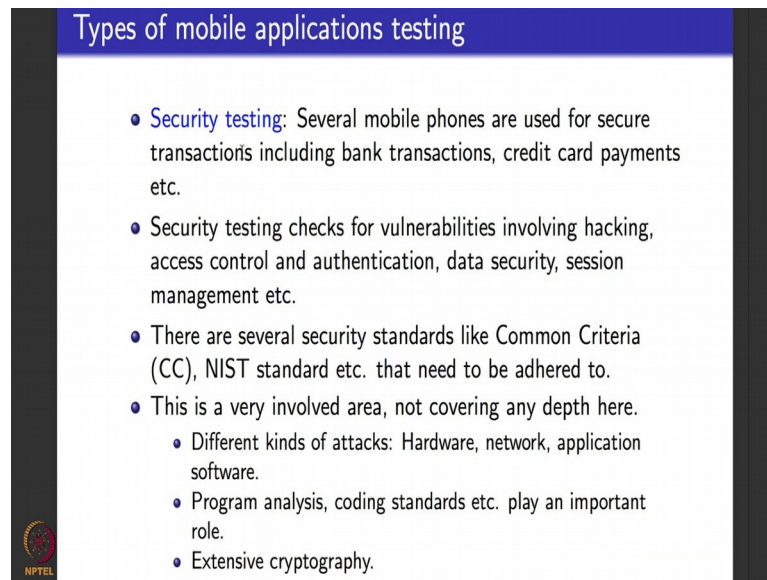
The slide is titled "Types of mobile applications testing" in a blue header. It contains three bullet points: "Interrupt testing: An application may face interrupts while installing or while in use.", "Interrupts include" followed by a sub-bullet listing "Incoming and outgoing SMS and MMS, incoming and outgoing calls, incoming notifications, cable insertion and removal for data transfer, network outage and recovery, media Player on/off, device power cycle etc.", and "Interrupt testing tests if an application correctly goes into a suspended state and resumes afterwards." In the bottom right corner, there is a small video inset showing a man speaking. The NPTEL logo is in the bottom left corner.

- **Interrupt testing:** An application may face interrupts while installing or while in use.
- Interrupts include
 - Incoming and outgoing SMS and MMS, incoming and outgoing calls, incoming notifications, cable insertion and removal for data transfer, network outage and recovery, media Player on/off, device power cycle etc.
- Interrupt testing tests if an application correctly goes into a suspended state and resumes afterwards.

So, going on, interrupt testing is another kind of mobile application testing, what it does is that you know an application on an mobile app might face several interrupts while installing or while in use.

Remember when I told you about web based testing, I told you the interrupt that people do back and refresh buttons and so on. Similarly for mobile app what could be interrupts, there could be a sudden call or it could be a incoming call or you might decide to make an outgoing call or they could be a message, I could be an incoming notification, you might decide to remove the battery while you are running a particular app, you might decide to charge the phone while you are running a particular app, you might suddenly lose network so many other different problems could be thought of as interrupts. So, interesting basically consider the fact about a mobile app working correctly even if interrupts come, and in the worst case it test whether the mobile app correctly goes into suspend state and resumes afterwards.

(Refer Slide Time: 18:48)



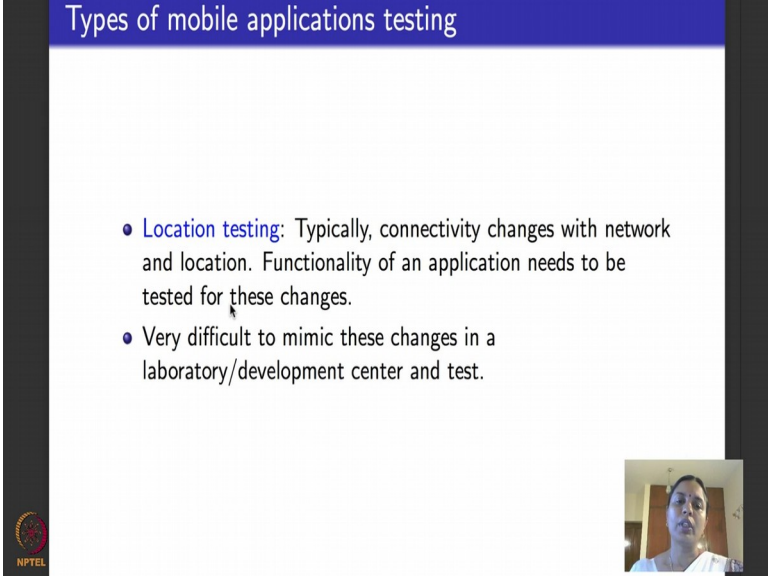
The slide is titled "Types of mobile applications testing" in a blue header. It contains a bulleted list of points about security testing. In the bottom left corner, there is a small NPTEL logo.

- **Security testing:** Several mobile phones are used for secure transactions including bank transactions, credit card payments etc.
- Security testing checks for vulnerabilities involving hacking, access control and authentication, data security, session management etc.
- There are several security standards like Common Criteria (CC), NIST standard etc. that need to be adhered to.
- This is a very involved area, not covering any depth here.
 - Different kinds of attacks: Hardware, network, application software.
 - Program analysis, coding standards etc. play an important role.
 - Extensive cryptography.

Security testing is very important for mobile apps, because we all a good number of us were smart phones not only use our phones to make calls and stay in touch with friends and family, but we also use it for several critical transactions like for example, we used for bank transactions to transfer amount to login securely into a bank account.

And very soon in India you can use your phone for making as a credit card, you can use it to make payment is already there in the USA. Securities a very important part of certain kinds of app; security testing checks for vulnerability is related to hacking access control, authentication, data security session management and so on. There are several security standards that this mobile apps that deal with these bank transactions and credit card payment and all have to meet, one of the common called CC expanded as a common criteria, NIST in the USA several standards, this is actually quite a deep area you could have half a course on security testing and the part of good part of it could be on mobile app security testing. I am not going into the details I am just giving you a very very high level overview on just by telling you that this is an important area the needs to be considered.

(Refer Slide Time: 20:22)



Types of mobile applications testing

- **Location testing:** Typically, connectivity changes with network and location. Functionality of an application needs to be tested for these changes.
- Very difficult to mimic these changes in a laboratory/development center and test.

NPTEL



In the part of this weeks of lecture I will tell you little more details about security and performance testing that might be helpful, but by no means we are going in any depth these area please remember. So, the last one I would like to discuss about what is called location testing, which is again very specific to mobile apps testing you would have typical experience the connectivity changes with network and location, suddenly you have no signal, suddenly the signal strength will be very good. So, you have to be able to test the functionality of a mobile app for these kinds of changes also. The problem with this kind of testing is very difficult while you are developing an app inside your organization, in it is very difficult to recreate this kind of scenario to test it right typically this happens when you are traveling right when you lose signal signals drop, signals to come back for various reasons.

If you are going to sit in your organization in your testing lab, imagine how difficult it be to recreate sudden drop of signals sudden picking up a single. So, it is one of the difficulties that people face while doing location testing is they may not be able to accurately mimic the change in location of network or a signal, within the laboratory or development center at test.

(Refer Slide Time: 21:28)

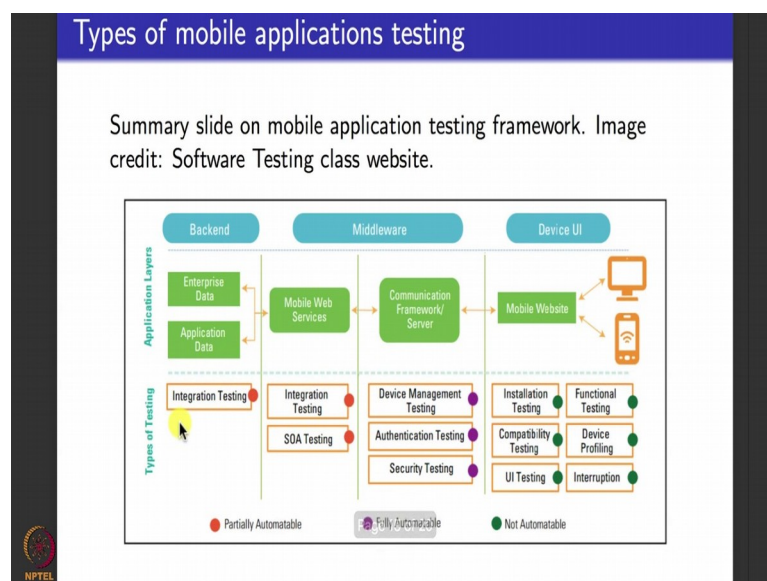
Types of mobile applications testing

- Apart from the listed system level features, normal unit testing, integration testing etc., are performed for mobile phone apps.
- These can be done using the criteria that we learnt in the course and using mutation testing.
- The listed system level testing are manually done, cannot be fully automated.



So, apart from the various system level features that we saw till now as I told you to do the normal unit testing, integration testing, everything it is just another piece of software, that can be done using any of the criteria that we learnt earlier in the course. And what we looked at in the just past few minutes are system level testing features specific to mobile apps. So, here is a summary slide that I picked up from another website called software testing class, I thought I show you this slide because it gives a nice overview of the current state of the art mobile app.

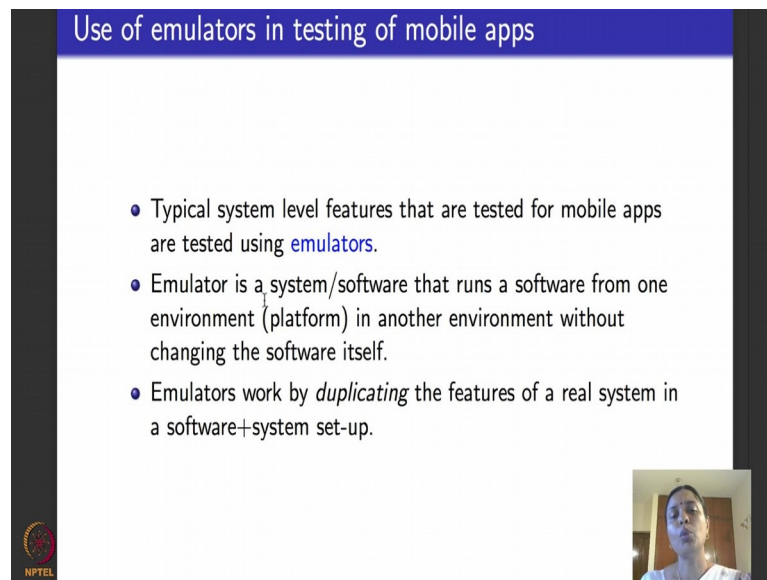
(Refer Slide Time: 21:52)



So, what are we will just focus on this part first it says types of testing. So, at the backend at the middleware or at the device user interface; backend of a mobile device middleware in the mobile device or at the user interface. At the backend when I am developing an app I do integration testing. I do integration testing at the middleware level also. At the middle level I also do authentication security testing, and device management testing, which talks about you know releasing keeping memory in tag releasing system hardware resources that you do not need and so on. But good number of system level testing happens at the device UI level, which is what we talked about installation, compatibility, user interface, functional testing, profiling of a device interruption all these things happen at the system level face and as this legend shows unfortunately good number of these cannot be automated.

So, it heavily relies on a person having knowledge about the mobile development environment, about the domain of the application about the mobile OS and then actually writing test cases to be able to pull out errors. The rest of it that these kind of stuff are heavily automatable there are tools, I will tell you about some of the tools that are available in open source.

(Refer Slide Time: 23:28)



The slide is titled "Use of emulators in testing of mobile apps" in a blue header. It contains a bulleted list of three points. In the bottom right corner, there is a small video inset showing a person speaking. The NPTEL logo is in the bottom left corner of the slide area.

- Typical system level features that are tested for mobile apps are tested using **emulators**.
- Emulator is a system/software that runs a software from one environment (platform) in another environment without changing the software itself.
- Emulators work by *duplicating* the features of a real system in a software+system set-up.

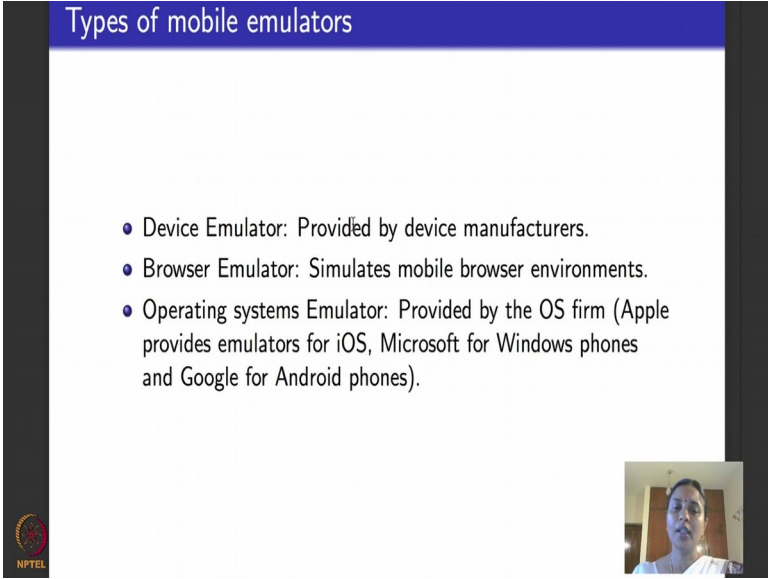
So, moving on to tools for testing mobile apps, as I told you very difficult to buy every kind of device available in the market do go around carrying your device waiting for network to become low battery drain out, we cannot do this things in a real environment.

So, what we do is we create simulated environments where these scenarios as depicted for a mobile phone, and a mobile phone functionality is tested. So, typically we used what are called emulators to do this, system level features that features that we discussed for mobile apps and tested using what are called emulators.

What is a emulator? An emulator can be thought of as a system, that again contains the piece software what is it do? It runs a software from one environment in another environment without changing the software. Like for example, you know using an emulator you could say you configure it for this particular device, that runs android operating system with this as the screen resolution and this as the display brightness, and this is how the parameters that touch screen coordinates in my screen looks like and then for the same app let say you want to test for I phone. So, you say you configure it for an iOS which is for this particular device; I phone device which has this screen resolution this kind of a display and so on. Emulator let us you do this configurations of a particular system software and hardware.

Such that you can run your app as if you are actually running it on the real phone, and test all the features of your app that you want to test. How do they work? They work by duplicating the features of a real system in a software plus system setup.

(Refer Slide Time: 25:19)



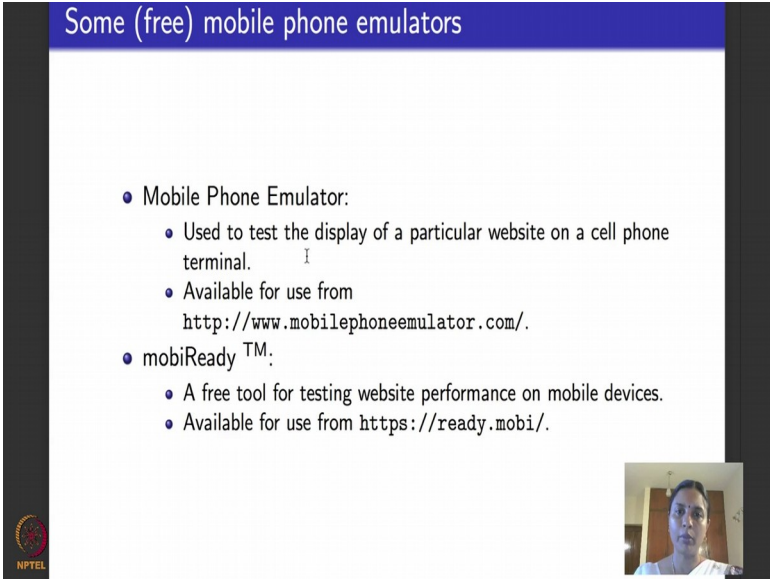
The slide is titled "Types of mobile emulators" in a blue header. It contains a bulleted list of three types of emulators. In the bottom right corner, there is a small video inset showing a man speaking. The NPTEL logo is in the bottom left corner.

- Device Emulator: Provided by device manufacturers.
- Browser Emulator: Simulates mobile browser environments.
- Operating systems Emulator: Provided by the OS firm (Apple provides emulators for iOS, Microsoft for Windows phones and Google for Android phones).

So, here are three kind of mobile emulator that are commonly known; first one is what called device emulator, this is typically provided by the device manufacturers themselves

if you want to develop an app or test an app. The second is browser emulator which emulates the browser settings available in a particular mobile phone, they as I told you it simulates mobile browser environments. Third is OS emulator which is again provided by particular firm that owns that OS apple provided for iOS Microsoft provided for windows phones, Google were provided for android phones so on.

(Refer Slide Time: 25:58)



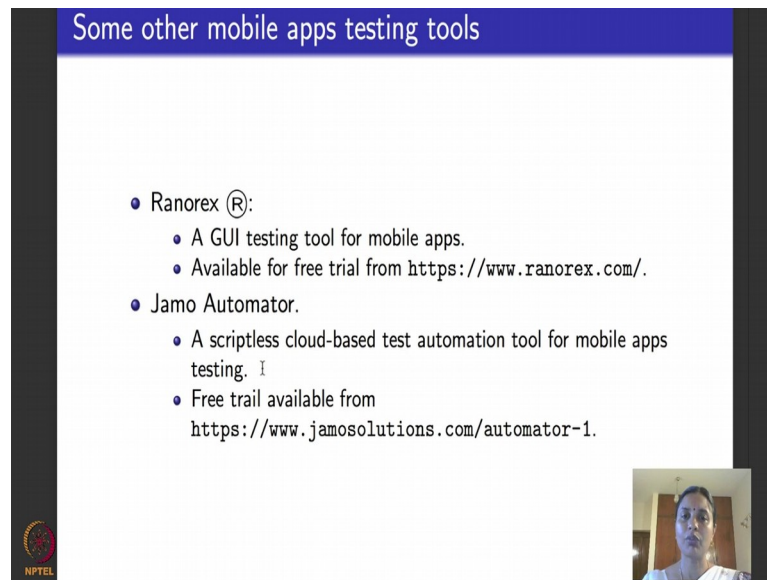
The slide is titled "Some (free) mobile phone emulators" in a blue header. It contains two main bullet points. The first is "Mobile Phone Emulator:", which has two sub-bullets: "Used to test the display of a particular website on a cell phone terminal." and "Available for use from <http://www.mobilephoneemulator.com/>.". The second main bullet point is "mobiReady™:", which has two sub-bullets: "A free tool for testing website performance on mobile devices." and "Available for use from <https://ready.mobi/>". In the bottom right corner of the slide, there is a small video inset showing a man speaking. In the bottom left corner, there is an NPTEL logo.

- Mobile Phone Emulator:
 - Used to test the display of a particular website on a cell phone terminal.
 - Available for use from <http://www.mobilephoneemulator.com/>.
- mobiReady™:
 - A free tool for testing website performance on mobile devices.
 - Available for use from <https://ready.mobi/>.

So, here are some free mobile emulators that I am aware of.

The first one is called mobile phone emulator, you can download it from this site, it is used to test the display of a particular website on a cell phone terminal right. As I told you suppose you have an bank thing bank Citibank online mobile app, that is available the way it is displayed on a iphone will be different from the way it is displayed on a Samsung phone. So, this emulator can be used to configure the display for these different different once, see how the particular display of a website looks like. Another emulator is mob ready, this a free tool for testing performance of a website on a mobile app you can pick it up from this site.

(Refer Slide Time: 26:47)



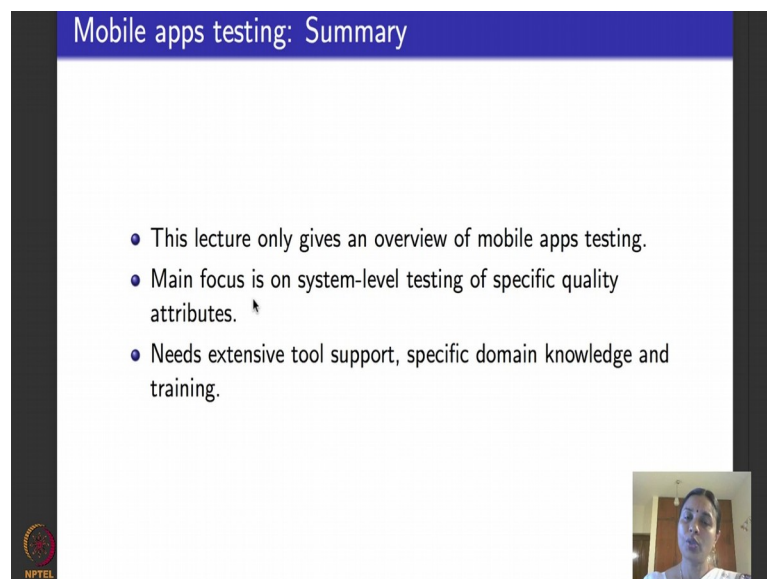
The slide is titled "Some other mobile apps testing tools" in a blue header. It contains a bulleted list of two tools: Ranorex and Jamo Automator. Ranorex is described as a GUI testing tool for mobile apps, available for a free trial from a specific URL. Jamo Automator is described as a scriptless cloud-based test automation tool for mobile apps testing, also available for a free trial from a specific URL. The slide includes an NPTEL logo in the bottom left and a small video inset of the presenter in the bottom right.

- Ranorex (®):
 - A GUI testing tool for mobile apps.
 - Available for free trial from <https://www.ranorex.com/>.
- Jamo Automator.
 - A scriptless cloud-based test automation tool for mobile apps testing. I
 - Free trail available from <https://www.jamosolutions.com/automator-1>.

Then there is another tool called ranorex which provides g u i testing tool for mobile apps, this is not an open source tool you can download it for free time from this website.

Another tool that I am aware of is jamo automator which is again not an open source tool, but there are free trials available, this is script less cloud based mobile test automation tool.

(Refer Slide Time: 27:08)



The slide is titled "Mobile apps testing: Summary" in a blue header. It contains a bulleted list of three points: the lecture gives an overview of mobile apps testing, the main focus is on system-level testing of specific quality attributes, and it needs extensive tool support, specific domain knowledge and training. The slide includes an NPTEL logo in the bottom left and a small video inset of the presenter in the bottom right.

- This lecture only gives an overview of mobile apps testing.
- Main focus is on system-level testing of specific quality attributes.
- Needs extensive tool support, specific domain knowledge and training.

So, just wanted to summarize this lecture what I intended to tell you in this lecture in response to some of your request in the forum, was to give an overview mobile apps

testing I am I know means an expert in this area. So, I managed to give you an overview that includes important system level features that are tested for mobile apps. Please remember that it need extensive tools of specific domain knowledge and testing to be able to test for mobile apps.

Thank you.

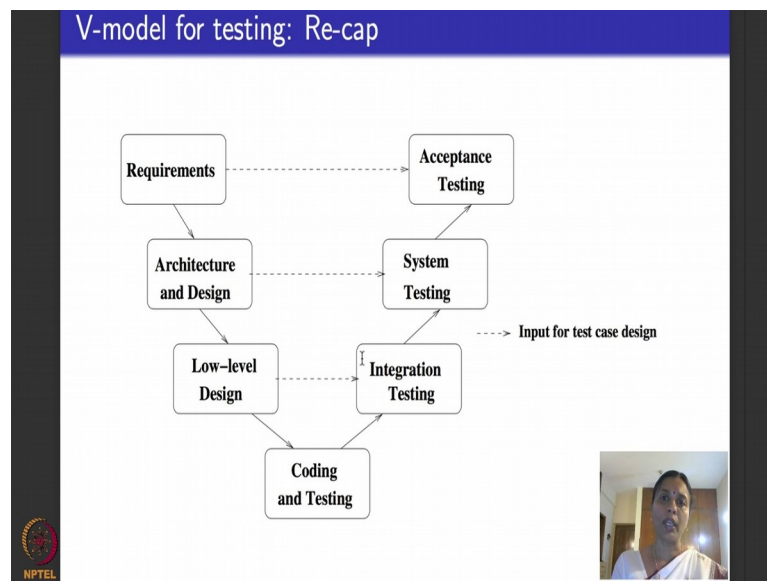
Software testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 57
Non-functional System Testing

Hello again, welcome to the last weeks next lecture. Again as I told you during the mobile apps testing lecture, in response to my pinging on the forum about what you would like to have, a few of you had also requested for a module on performance testing. So, here is a module that caters to that.

This particular lecture does not do only performance testing because either you do it by opening a tool, some of them proprietary and cater to a web application or you give an overview. If I give an overview, then it would not be enough for one lecture, but if I open a tool and then goes out of the focus of this course, which was focusing on algorithms. So, what I decided I will do is that I will give you on a overview of all nonfunctional, popular nonfunctional system testing techniques and performance testing will come as a list in this set of techniques. So, that is what I thought I will cover. Next module I will do regression testing. So, here is an overview what we are going to do in this lecture.

(Refer Slide Time: 01:38)



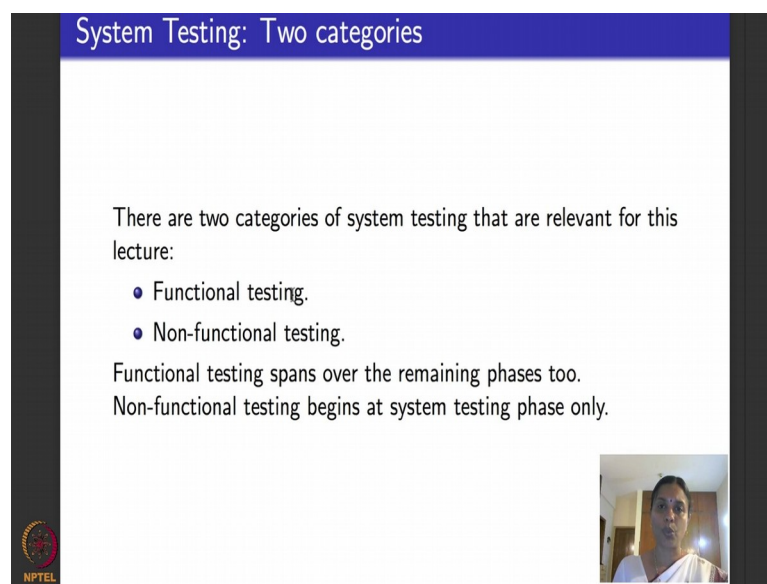
I will give you a taxonomy of system testing that is typically done and we will focus on as I told you non-functional system testing and as requested, I will also look into a little

more details about performance testing by giving you tools that are open source proprietary, what are the goals to measure and so on. So, if you remember, I had shown you this V model long ago when we began in the first week in the course. So, on the left hand side of the V model is what we have, the normal software development life cycle. We do requirements architectural design, coding, low level design, coding and then we do testing. But when it comes to focus testing, we usually expand the V model into its right arm which tells you what is the kind of testing that is done.

Right here at the bottom, along with coding, I do unit testing. In an agile methodology typically done by the developer, after which I do integration testing, software modules are an integrated, software is integrated with hardware database and tested. Then we do system level testing where everything is put together with the entire working system in test it finally, acceptance testing. This model we had looked at long ago. What did we focus on in the course, till the past few weeks, they were here and here, coding unit testing and integration testing. What is be in the focus of this week and in the past when we did web applications and object oriented testing, that has been system level testing. Today again will look at system level testing, but not focus on functional system level testing instead focus on what is called non-functional system level testing.

Again in the first week in my introductory lectures, I had introduced you to 2 kinds of testing, several dichotomies testing and one of it was functional and non-functional.

(Refer Slide Time: 03:14)




System Testing: Two categories

There are two categories of system testing that are relevant for this lecture:

- Functional testing.
- Non-functional testing.

Functional testing spans over the remaining phases too.
Non-functional testing begins at system testing phase only.

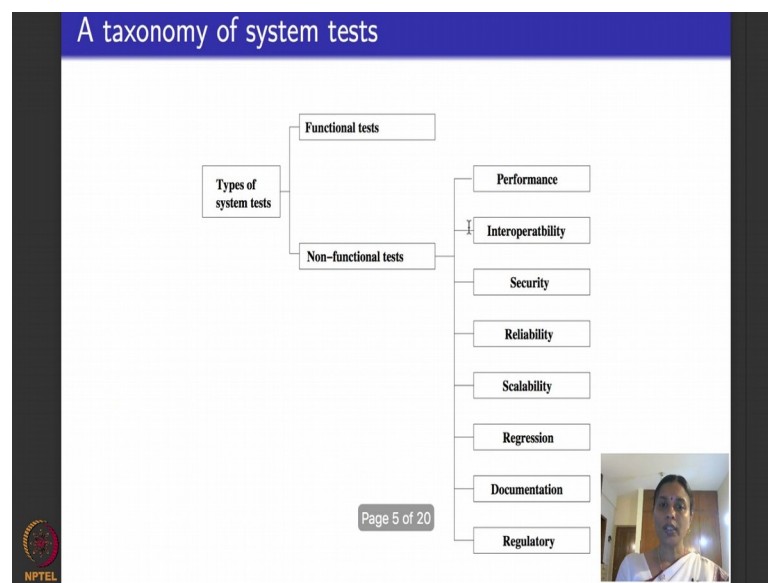
NPTEL



So, here is the category that we did, right in the first week that we are recapping again, there are 2 category of system level testing that we would like to recap. One is called functional testing which we have looked out majority of the course, basically tests if the software needs its requirement, white box, black box.

Anyway, the next is called non-functional testing which basically tests the software to check if it needs various quality attributes. As per ISO standard, there are 64 different quality attributes most of them ending with the word “LITY” or TY. So, many organizations popularly abbreviate them and call them as “ilities” which is what nonfunctional testing focuses on. Non-functional testing typically begins only at the system level. We do not really do non-functional testing, here at coding and unit testing or at integration testing, but the work for non-functional testing, as we saw in the V model begins along with requirements architecture and design. Typically, it is the architecture that caters to quality parameters the work in terms of what to test for, input for test case design, they get ready here actual testing happens here.

(Refer Slide Time: 04:29)

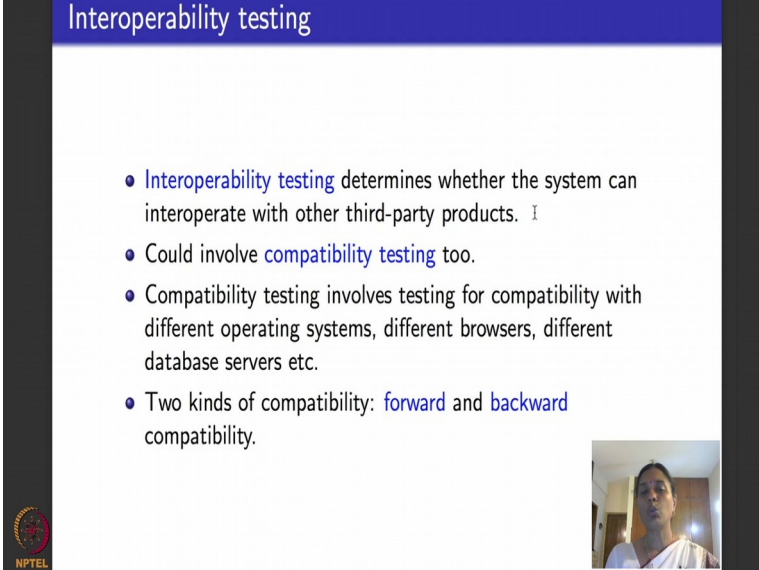


So, our focus is on nonfunctional testing. So, here is the same dichotomy which expands into nonfunctional tests. So far, our classification, there are 2 kinds of system tests: functional system test and nonfunctional system test. There is several different varieties of nonfunctional system test, I have chosen to put a few popular among them and I

should also explicitly tell you that I have left out a good number of them. So, this by no means is an exhaustive listing of all the non-functional tests.

What we will be doing in this lecture is sort of go through them one after the other. So, performance, I will reserve for the end. We will begin with interoperability, follow it with a brief overview on security and reliability. To iterate what I told you in the context of mobile apps testing, this to a very involved area. This course what we do is by no means even a cursory introduction to these elaborate testing fields, we look at what is scalability testing for scalability regression testing, I will do a separate half an hour module because again, there was a request for that today I will also tell you about documentation and regulator testing. So, system test 2 categories functional and nonfunctional are focused today is nonfunctional test and specifically re-look at all this except for regression testing which will do in a separate module, and security and reliability need in depth analysis which we will not be able to do in the course due to lack of time. So, rest I will tell you. We begin with interoperability testing.

(Refer Slide Time: 06:03)



The slide is titled "Interoperability testing" in a blue header bar. Below the title, there is a bulleted list of four points. The first point states that interoperability testing determines if a system can work with other third-party products. The second point mentions that it could involve compatibility testing. The third point explains that compatibility testing involves testing for compatibility with different operating systems, browsers, and database servers. The fourth point notes that there are two kinds of compatibility: forward and backward. In the bottom right corner of the slide, there is a small video inset showing a person speaking. In the bottom left corner, there is a small NPTEL logo.

- **Interoperability testing** determines whether the system can interoperate with other third-party products. I
- Could involve **compatibility testing** too.
- Compatibility testing involves testing for compatibility with different operating systems, different browsers, different database servers etc.
- Two kinds of compatibility: **forward** and **backward** compatibility.

So, what is the word interoperability tell you or remind you of? It says you something should work with the other across platforms; across operating systems, across languages. So, that is what interoperability testing does. It determines whether the system can operate or interoperate along with other third party products.

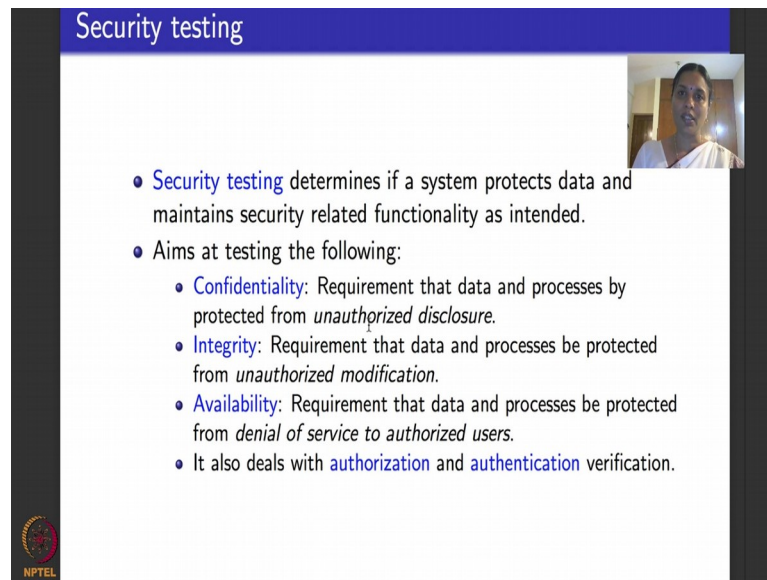
Suppose I say that I have recording software and it should be able to operate with any kind of camera on any system. It could be installed for a MAC system for a windows system, or for a Linux system, but whatever it is, let us say there is there is an inbuilt camera, there is an externally fixed camera there is a USB camera port whatever it is my recording software should be able to pick up the data from the camera and record. So, it should be interoperable along with the third party software or a hardware component. That is what interoperability testing does. A particular wing interoperability testing is what is called compatibility testing.

What is compatibility testing do? It involves checking for compatibility with different operating systems, different browsers, different database servers and so on. When we did the module on mobile apps testing, I had spoken to you a bit about this. So, when I say I have an app, a good enough app, I make it amenable to run on an Android OS or an IOS and so on. Similarly I have piece of something, let us say a tool like Junit, will it run on Windows machine, will it run on Linux machine, that is what compatibility testing does.

Compatibility testing also checks for forward and backward compatibility. Let me you give an example for that let us take you, let us say you have Microsoft office software which has Microsoft office tools suite, power point, Microsoft office word, excel, outlook. So, many other things right and they keep releasing newer and newer versions. So, now, let us say you have an office word document that you open with an older version, let us say 2007, it should be compatible with the newer version which ever, in the sense that if I had developed document with an older version, I should be able to open and edit it in a newer version. That is what is call forward compatibility.

Backward compatibility means something was developed in a later version then I should be able to at least open it in the previous version, may not be able to edit it if it uses features of newer features of the later version, but I should at least be able to open it and view it. So, interoperability also deals with checking for compatibility. Sometimes you will get messages, especially when you are working with some websites operated by governments banks saying that this is compatible only with the following browsers and the following versions of these browsers which means what it has features that suit only these browsers and these versions. It is not compatible with other browsers and their versions that are not specified in the list. So, this is another important non-functional testing that people do at a system level testing for software or a system.

(Refer Slide Time: 09:29)



The slide is titled "Security testing" in a blue header. It contains a list of bullet points explaining the concept. In the top right corner, there is a small video inset showing a person. In the bottom left corner, there is a small NPTEL logo.

- **Security testing** determines if a system protects data and maintains security related functionality as intended.
- Aims at testing the following:
 - **Confidentiality**: Requirement that data and processes be protected from *unauthorized disclosure*.
 - **Integrity**: Requirement that data and processes be protected from *unauthorized modification*.
 - **Availability**: Requirement that data and processes be protected from *denial of service to authorized users*.
- It also deals with **authorization** and **authentication** verification.

So, the next security testing. Just to repeat that the cost of being repetitive, I had tell you that we are only scratching the surface, this is very involved area feel free to browse through other courses or books that talk about it, I am just giving you the introduction to that term. So, it determines if a system or a product protects the data it handles with lot of data like banks means our account data that sensitive and private to us, user name password and so on, and it does it maintained security related functionality as it was intended to be. Security testing involves further “ilities”: confidentiality, integrity and availability.

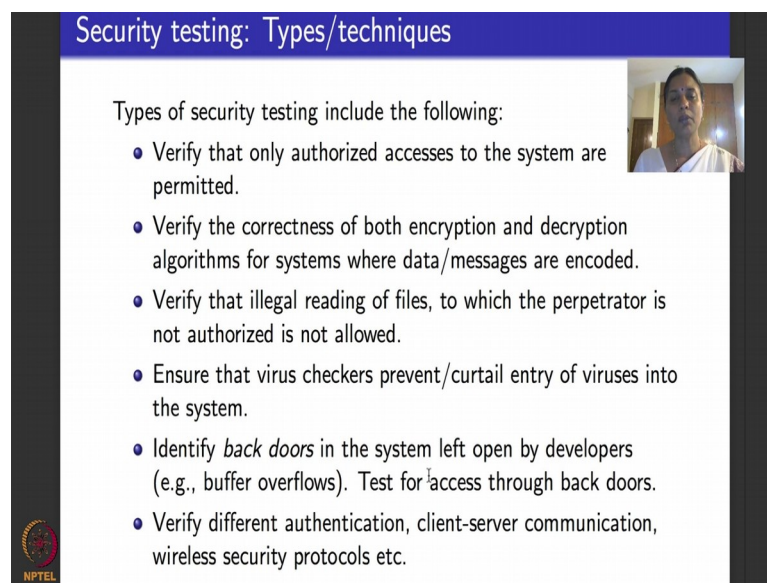
What is confidentiality; say as the word indicates, it is a requirement, it tests for requirement that data and processes set up by a software or a system, is protected from unauthorized disclosure. I get my bank information provided I have specified the credentials correctly. I get my system marks on-line provided, it is me, I have provided the user name and password correctly; that is confidentiality. Integrity is tests for requirement the data and processes be protected from unauthorized modification. Somebody manages to get access to my data, person is an authorized to access my data then the person continues to be unauthorized to modify my data because that is even bigger than accessing the data, that is integrity testing.

The next one is availability testing. It tests that data and processes be protected from denial of service to authorized users, means if a particular authorized person is

authorized to have access to a particular data or a service, then the data or the service is made available to user. So, that is what availability testing. So, security testing tests for confidentiality, integrity, availability, it also deals with authorization and authentication validation. Authorization to give you an example, typically you might say that if something is available in the private mode then only the person who owns it is authorized to view it, but something is made public then everybody is authorized to view it. So, it gives permissions to save who is authorized to view access read or write to certain content.

Authentication is the process of validating a particular user. There are authorization policy is like role based authorization, MAC, DAC and so on that are popular across system software provided by several different products and typically security testing involves validating for authorization and authentication policies also.

(Refer Slide Time: 12:16)



The slide is titled "Security testing: Types/techniques" in a blue header. Below the title, it says "Types of security testing include the following:" followed by a bulleted list of seven items. In the top right corner of the slide content area, there is a small video inset showing a man speaking. In the bottom left corner, there is a small NPTEL logo.

Security testing: Types/techniques

Types of security testing include the following:

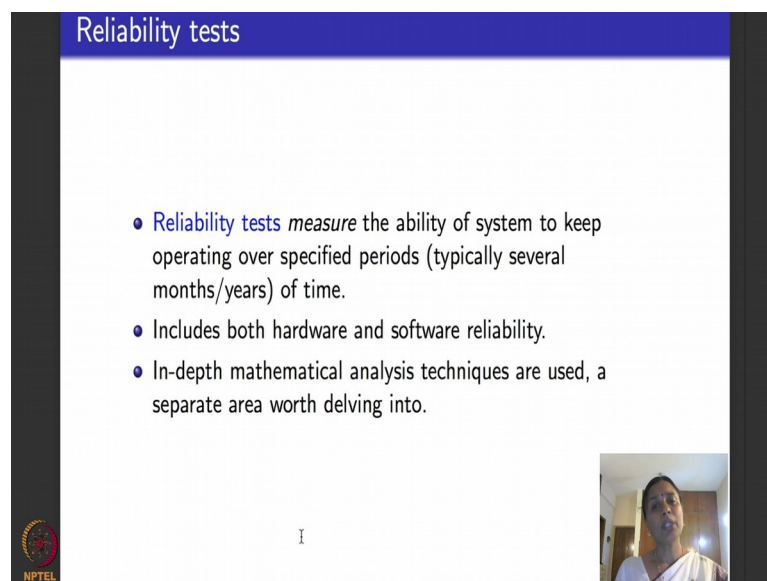
- Verify that only authorized accesses to the system are permitted.
- Verify the correctness of both encryption and decryption algorithms for systems where data/messages are encoded.
- Verify that illegal reading of files, to which the perpetrator is not authorized is not allowed.
- Ensure that virus checkers prevent/curtail entry of viruses into the system.
- Identify *back doors* in the system left open by developers (e.g., buffer overflows). Test for access through back doors.
- Verify different authentication, client-server communication, wireless security protocols etc.

So, the various types of security testing will test for the following. We will verify that only authorized access to the system are permitted. It will also verify that the encryption and decryption algorithms that are used for encoding data or messages work correctly. It also verifies that illegal reading of files to which a particular user is not authorized is actually not allowed. It ensures that if you have an anti-virus software, they actually prevent, far from the preventing they also curtail entry of viruses into the system.

Typically security testing, also testing tools and techniques also identify what are called back doors in the system.

Back doors in the system referred to classical coding mistakes that developers could make which provide illegal access to an unauthorized user or hacker. One of the back doors is a buffer overflow which hackers supposedly use to use to hack into a system. So, security testing also systematically works towards identifying such back doors and ensures that access through these back doors actually not possible. As I told you, it verifies for authentication, authorization and there are several protocols at various layers that these security policies use. Based on what their communication interfaces, they could use several wireless security protocols, client server security protocols, authentication protocols. Each of them has to be separately tested at appropriate different layers security testing covers for all of them also.

(Refer Slide Time: 13:57)



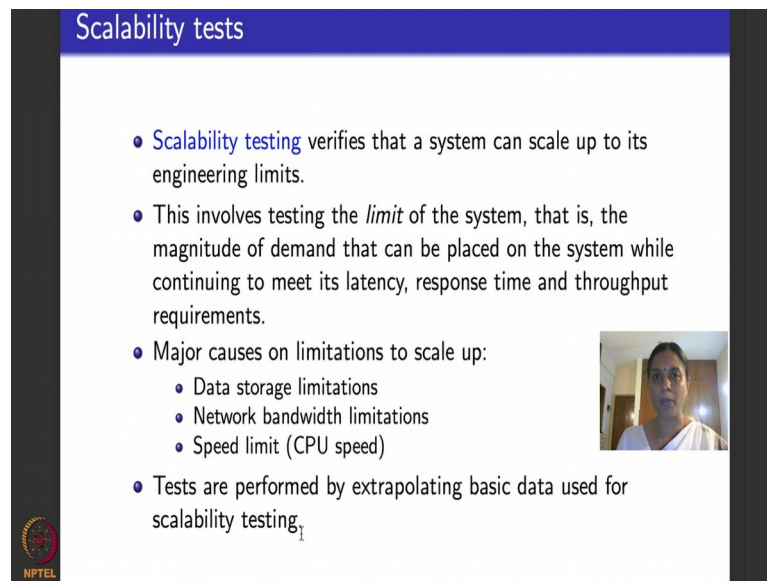
The slide is titled "Reliability tests" in a blue header. It contains three bullet points: "Reliability tests measure the ability of system to keep operating over specified periods (typically several months/years) of time.", "Includes both hardware and software reliability.", and "In-depth mathematical analysis techniques are used, a separate area worth delving into." In the bottom right corner, there is a small video inset showing a man speaking. The NPTEL logo is in the bottom left corner.

- Reliability tests measure the ability of system to keep operating over specified periods (typically several months/years) of time.
- Includes both hardware and software reliability.
- In-depth mathematical analysis techniques are used, a separate area worth delving into.

Next is reliability. The term reliability indicates that the system is reliable in the sense that if it runs for really really long time, it continues to perform, does not degrade. Let us say I buy a particular electrical appliance, what does it mean for an electrical appliance to be reliable? Let us say if I buy a fan, what I mean it to be reliable; I expect it to work for many years without failing, that is what I mean for a system to be reliable. So, what is reliability testing do its test to measure the ability of a system or software to keep operating over a specific period of time. This specific period of time is usually long


measured in terms of years or sometime months. Reliability includes hardware reliability and software reliability. It involves different techniques. Both of them deploy a lot of probability theory and Statistics which we have not covered in this course. So, this is a separate area that you can run a separate course on, there that much of information and literature available.


(Refer Slide Time: 14:58)



Scalability tests

- **Scalability testing** verifies that a system can scale up to its engineering limits.
- This involves testing the *limit* of the system, that is, the magnitude of demand that can be placed on the system while continuing to meet its latency, response time and throughput requirements.
- Major causes on limitations to scale up:
 - Data storage limitations
 - Network bandwidth limitations
 - Speed limit (CPU speed)
- Tests are performed by extrapolating basic data used for scalability testing.





So, the next in a list of nonfunctional system level testing would be scalability tests. What does it do? It says will the system scale in the sense that let us say a particular server or a online web software lets say HDFC bank online, it is supposed to cater to 100-1000 simultaneous users logging in and accessing the respective account information. Another classical example is our Indian railways booking website, right, is it scalable. The normal load could be several 1000 users during Tatkal, it could be get higher number. Does it cater to the maximum number of users that it is supposed to cater to? Can its scale up to its limits? The limits need not be only in terms of users, it could be in terms of any engineering parameter that we specify. For example, one is the number of users, the other is the size of the database that it can manipulate, like for example, if you take an Aadhar system, right, Aadhar system should be scalable to cater to our country's population. So, that is it engineering limit, right.

So, scalability testing involves testing the limit of a system that is it is the magnitude of demand that can be placed on the system as it continues to perform. Perform in the sense

that it has a committed response time, no matter if there are ten users logged into the Indian railways ticket booking website or if there are 100 users or 1000 users logging into Indian railways ticket booking website, it still meets its commitment of booking tickets upon availability within a specified time. So, that is what scalability testing does. So, what could be the causes that can limit a system from being not being scalable. Typically, it could be the typical major causes that prevents the system from being scalable are, the data storage could be limited, in the sense of accessing the database and actually storing the data. It is a very big problem actually to be able to store data in such a way that it get it does not get heated up, especially for large systems like Facebook and Whatsapp and all data storage is a huge problem or even Google and any things like that.



The other big problem that could limit scalability is network bandwidth. It could be less, it should work, it should switch over to non GUI graphics, plain html kind of thing. Next limiting factor could be the speed limit, by speed limit, I actually here mean the CPU speed; the processor speed. So, scalability tests, actually what they do is that it is very difficult to create use case for a maximum number of users. Let us say a system says it can cater to up to 500000 users, it is very difficult to actually write a test case that will create a scenario of 500000 users logging to a system and accessing data. Such kind of test cases are very expensive to create.

So, what they do is that they create test cases to the extent that they can and then they extrapolate the basic data and test the system for scalability, that is how scalability testing works. So, it again needs a lot of statistics.

(Refer Slide Time: 18:15)

Documentation testing

- **Documentation testing** is done to verify the technical accuracy and readability of various documentation including user manuals, tutorials, on-line help etc.
- Usually performed at three levels:
 - **Read test:** A documentation is reviewed for clarity, organization, flow and accuracy without executing the documented instructions on the system.
 - **Hands-on test:** On-line help is exercised and the error messages verified to evaluate their accuracy and usefulness.
 - **Functional test:** Instructions embodied in the documentation are followed to verify that the system works as it has been documented.



So, the next kind of non functional system testing that was there in my list is documentation testing. So, basically whenever you have a software or a product or a system, it comes with several documentations, like a user manual which could be an online manual or a printed manual, it could have tutorials, it could have an online help, lot of these kind of things are there. So, they constitute what is called the documentation of software, apart from the actual code also being documented.

So, documentation testing is basically checking if all this documentation is correctly done to the extent that it is supposed to be. So, it is done to verify the technical accuracy, it is correct or wrong technically and the readability, is it readable, is it understandable, of the various documentations. Such kind of documentation includes user manuals tutorials online help and so on. Documentation testing is usually performed at 3 different levels the first level is what is called a read test.

Basically somebody sits, competent in the area, sits and reads the documentation. In that process, the documentation is reviewed for clarity, for organization in the sense, the contents, are they laid out correctly, a chapter is organized properly, sections are organized properly. The flow and the accuracy, this is just static documentation testing nothing is executed. Then there is something called hands on documentation testing where an online help is actually exercised by a user, like a dummy login is created and then they go as if they do not know anything, look for online help, see if it provides the right kind of help, see if it provides the right kind of error messages and then they sit and evaluate how useful it is, how accurate it is.

Then there is something called functional documentation test which basically checks of the instructions that are given in the documentation are followed and verifies so that the system works as it is supposed to be. It means its functionality as documented. So, some of the tests that people usually do while doing documentation testing are as follows. Somebody as I told you manually reads through all the documentation to check if the grammar is correct, there are no typographical errors, technical terms has been consistently used and so on. Somebody reviews the use of graphics images like for example, you know typically, let us say you are buying a product like a car, then there could be several views of a car, in certain view you can see certain parts of a car.

So, it is the graphics giving the correct view of a car, correct view of the cross section of a car, of the front of a car, of the chassis of a car and so on, that is what people check for graphics and images. Then somebody actually verifies the glossary of terms that a company, the documentation and somebody verifies that there is a proper indexing of terms that is available as a part of the user manual. For every index, as you typically see, there will be a term and a page number where the term appears. So, they go and manually check if the page numbers have been correctly generated. Most documentation generated, software can do this automatically and the scope for error is less because it is automatically done.

But people typically still go and check. This is a part of their routine documentation checks. Then they go and verify that the if you have a printed version and an online version of the user manual they are actually one and the same; they are not different, they do not differ from each other, then they verify if the specified installation procedure is actually working correct by executing the steps on the actual system, then they verify the troubleshooting guide with the actual scenarios. So, this is all, it is quite exhaustive, they do a lot of exhaustive documentation testing and it is a pretty non trivial non-functional testing.

(Refer Slide Time: 22:15)

Regulatory testing

- Each country has regulatory bodies guiding the availability of a product in that country.
- CE (Conformite Europeene), CSA (Canadian Standards Association), FCC (Federal Communications Commission) etc. are some of them.
- In addition, safety critical systems have their own regulatory requirements, per, domain.
 - Aerospace: ARP standards, DO standards by RTCA etc.
 - Automotive: IEC 61508, MISRA guidelines etc.
- Exhaustive testing and documentation is done throughout the development to cater to these regulatory standards.



So, the next in a list is regulatory testing. What do we mean by regulatory testing? If you are bought a toy, you know typically you would see that it has these words called CE printed in a little sans serif style. So, that stands for Conformity European. Sometimes you would see FCCs, these days you see FSSAI in food products. So, these are all regulatory standards that describe or tell you that the particular product including software operating on a system, is of a certain quality. So, to be able to get that kind of regulatory assessment passed, usually software and system developers have to specifically undertake a testing called regulatory testing. They basically insist that you do a huge amount of documentation and perform routine tests as mandated by the corresponding regulatory authority.

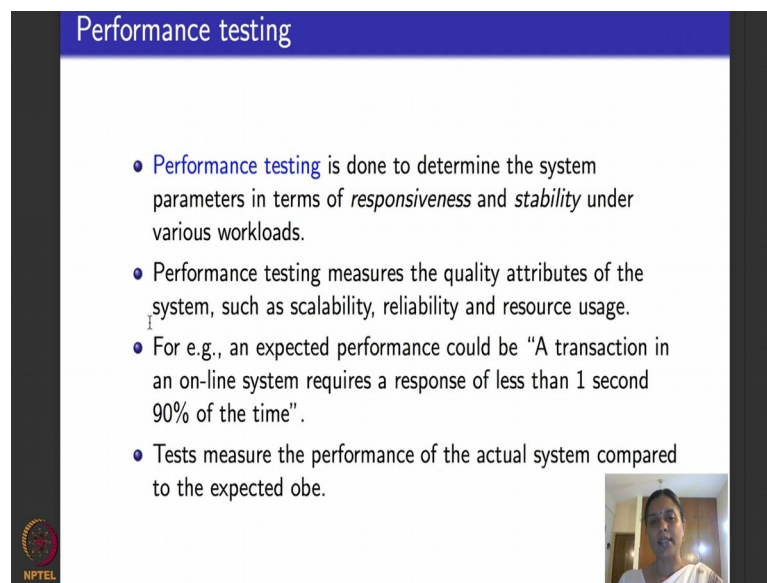
In addition to this, if you have safety critical systems, like for example, the drive by wire, brake by wire, steer by wire or the power window controller and a car, the ABS system, the airbag system or let us say autopilot of a airplane, the landing system of an airplane, these are safety critical because if they fail then they mean huge losses in terms of loss of life.

So, safety critical systems have to go through extra regulatory requirements. I had told you about this on and off in my lectures about this, There is this particular standard called DO standards for avionics. Similarly for automotive there is IEC 61508, Misra guidelines and so on. In fact, latest you might have heard of this Google and Tesla's driverless cars. So, they have even stricter regulatory standard. So, basically people do

regulatory testing to cater to a particular standardization based on the domain of applicability of the software and the system.

So, now we finally, get into performance testing. So, what is performance testing do? It is testing done to determine if the system parameters in terms of responsiveness and stability work fine under various workloads.

(Refer Slide Time: 24:15)



The slide is titled "Performance testing" in a blue header. It contains a bulleted list of four points. The first point defines performance testing as determining system parameters like responsiveness and stability under various workloads. The second point lists quality attributes measured: scalability, reliability, and resource usage. The third point provides an example: "A transaction in an on-line system requires a response of less than 1 second 90% of the time". The fourth point states that tests compare actual system performance to expected objectives. In the bottom right corner, there is a small video inset showing a person speaking. An NPTEL logo is visible in the bottom left corner of the slide area.

- Performance testing is done to determine the system parameters in terms of *responsiveness* and *stability* under various workloads.
- Performance testing measures the quality attributes of the system, such as scalability, reliability and resource usage.
- For e.g., an expected performance could be "A transaction in an on-line system requires a response of less than 1 second 90% of the time".
- Tests measure the performance of the actual system compared to the expected obe.



So, performance testing measures the performance of the system for several different quality attributes. Typically it performs it for scalability, reliability, resource usage and a few others that we will see very soon. So, for example, here is a typical example of how a performance testing requirement could be. A transaction in a particular online system requires a response of less than one second, ninety percent of the time.

So, what does it say, it says that the system will respond fast enough. How fast, in less than 1 second, and not always not very little, but most of the time. The "most" is perfectly quantified it says 90 percent of the time. So, this is a typical example requirement of performance.

(Refer Slide Time: 25:28)

Performance testing techniques

- **Load testing:** It is the simplest form of testing conducted to understand the behaviour of the system under a specific load. Load testing will result in measuring important business critical transactions and load on the database, application server, etc., are monitored.
- **Stress testing:** It is performed to find the upper limit capacity of the system and also to determine how the system performs if the current load goes well above the expected maximum.



So, when a performance test for this requirement, I create data that check helps me to validate whether this requirement is met or not. So, typical performance testing techniques are classified as follows. These are the most popular ones, but again, they are not exhaustive, there are some more that I have not listed in these slides. Load testing is a performance testing technique. What does it do? It is in fact the simplest performance testing technique that is conducted to understand how a system behaves under a certain load. Like I told you, right, if I have IRCTC system a peak load, let us say for such a system occurs during Tatkal booking hours, load testing checks if the system actually caters to that peak load. It will result in measuring important business critical transactions, load on the database applications server and so on.

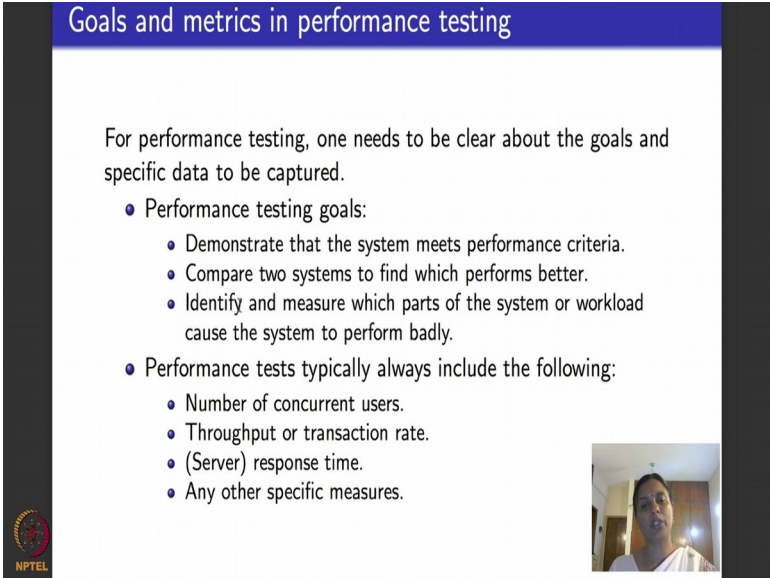
Stress testing is another kind of performance testing. It is performed to find the upper limit capacity of the system and also to determine how the system will perform if it is average current load goes well above the expected maximum. Let us say, there is a particular system. Let us say a system that displays the marks of the results of a higher secondary school examination. You can imagine that the system will be very stressed for the first few hours, after the results are out, right. So, what they do is that let us say it is supposed to cater to a 100000 students accessing their marks soon after the results are out, when this stress test it, they will test it for a limit above 100000 to see how the system actually responds to it.

They will stress the system in terms of giving inputs to the system and see if the system is able to cater to it or if it fails, how exactly does it fail. The next kind of performance

testing is what is called soak testing also known as endurance testing. This is performed to determine system parameters under continuous expected load. So, you understand the meaning of the word soak, right. Soak means what I keep it soaked, I see how, if there is a steady load that comes in continuously of a standard number of users periodically without a break, does the system perform as it is expected to?

So, that is what soak test does, typically, it is useful for how much memory is utilized and whether there are memory leaks or not. Then there is something called spike testing which is a kind of stress testing. So spike testing is like stress testing, but what it does is that it increases the number of users suddenly. There is a sudden spike, not a gradual spike, and see how well the system performs under the sudden increase in the number of users. The increase in the number of users need not be above the maximum number, but it is a fairly sudden increase. So, the main aim is to determine whether the system will be able to retain the sudden increase in the workload.

(Refer Slide Time: 28:29)




Goals and metrics in performance testing

For performance testing, one needs to be clear about the goals and specific data to be captured.

- Performance testing goals:
 - Demonstrate that the system meets performance criteria.
 - Compare two systems to find which performs better.
 - Identify and measure which parts of the system or workload cause the system to perform badly.
- Performance tests typically always include the following:
 - Number of concurrent users.
 - Throughput or transaction rate.
 - (Server) response time.
 - Any other specific measures.

NPTEL



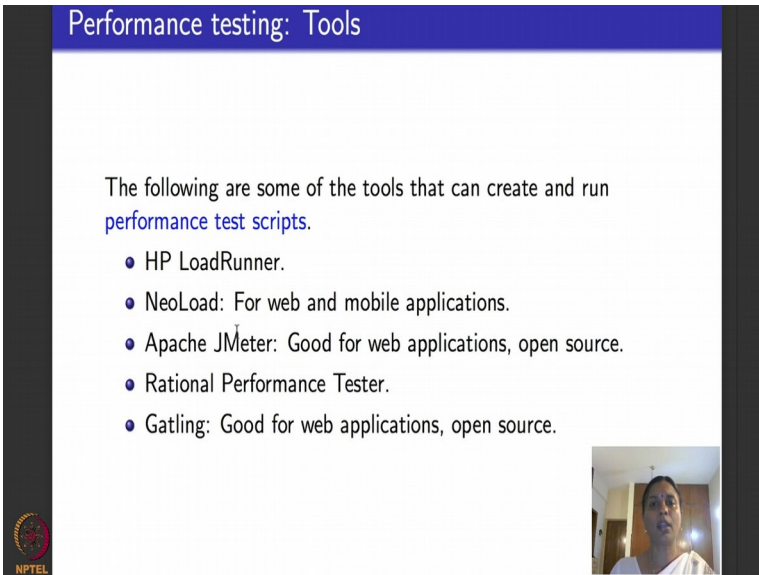
So, I have covered only 4, but there are several other performance tests as I told you which I have not really not mentioned or covered here. So, for performance testing, typically, it is important to know what is it that you are going to measure, what are your goals, what is it the data that I want to capture, what am I going to measure because if you go all over the place there are so many parameters you can capture, so much data

you can capture, typically as a tester, you will be left with an overdose of information which too much of data that cannot be analyzed.

So, it is important to find out what is the kind of performance testing that one is going to do, currently doing and for this kind of performance testing what is the precise goals I have, what is the set of data that I want to measure to be able to achieve this goals. So, typically, there are 3 kinds of performance testing goals that people can have. You could take a single system and demonstrate that the system meets its specified performance criteria. You could compare 2 different systems of the same kind and determine which performs better than the other with reference to a specified set of criteria or you could identify and measure the parts of a system or workload that causes a system to perform badly.

The violating parameters you could focus on that. So, typically whatever you do, performance tests will include one of these or most of these parameters. Let us say most of the systems are meant to interact with users. So, it will say how many number of concurrent or parallel users are using a system, that is one of the things. Then what is the throughput or transaction rate that the system is committing to and what is the server response time. In addition to these three, any other specific measures that you want to include can be included.

(Refer Slide Time: 30:13)



The slide is titled "Performance testing: Tools" in a blue header. The main content area is white and contains the following text: "The following are some of the tools that can create and run performance test scripts." followed by a bulleted list of tools. In the bottom right corner, there is a small video inset showing a person speaking. In the bottom left corner, there is a small NPTEL logo.

Performance testing: Tools

The following are some of the tools that can create and run performance test scripts.

- HP LoadRunner.
- NeoLoad: For web and mobile applications.
- Apache JMeter: Good for web applications, open source.
- Rational Performance Tester.
- Gatling: Good for web applications, open source.

NPTEL

So, here are some performance testing tools that are very popular? some of them are proprietary, some of them are open source, I have marked those that are open source against the tools, the rest of them are proprietary. The most popular performance testing tool in my opinion is that by HP, it is called load runner, that is what is best. Before I move on, what do these kinds of tests do?



They basically generate scripts for execution. They generate large number of performance scripts based on your target of performance measurement and they also execute these scripts that is the purpose of these tools. The popular one is by HP load runner been there for many years. There is one more tool called Neo load which is useful for web and mobile applications performance testing. There is a tool called JMeter which is good for web applications, this is an open source tool. There is another tool called rational performance tester again a proprietary tool there is a tool called Gatling which is again good for web applications and open source.

If you recap of my lectures on web applications, I had given you a link to another a web applications online testing tool. That tool is an exhaustive, that page is an exhaustive listing and includes several other performance testing tools also. Here I have just put some of the ones in particular HP LoadRunner and JMeter, I have tried out. The rest I have heard others using it. So, I thought I will put this list. So, typically after these performance testing tools generate and execute the scripts, you might have decided on the parameters that you want to monitor like the ones that I mentioned here.

(Refer Slide Time: 31:55)

Performance monitoring

- The system running the generated test scripts needs to be monitored to observe the intended performance parameters.
- All the decided parameters are monitored with help of special scripts or external devices hooked to the system in which the test scripts are running.



So, you; how do you monitor these parameters? You actually hook the system to external things, you are a separately run off loaded scripts or drivers that will monitor the system or even hardware devices that are plugged into monitor these parameters. They are monitored as I told you with the help of special, dedicated scripts that do not run as a part of the main system, but run separately so that the performance of the main system is not affected. Or they could even be monitored with the help of external devices that are hooked to the system in which the test scripts are running. So, that is the brief overview of performance testing and non-functional testing. I will go back to the slide that I began with the taxonomy slide. We did cover an overview of all of these I will do one separate lecture on regression testing. It also helps to know what I did not do before we end this module on non functional system testing

(Refer Slide Time: 32:46)

What we didn't look at?

We didn't cover the following topics:

- Usability testing
- Testing of GUI
- Security
- Reliability

There is fair amount of material available in the above areas, still active research is being done.

NPTEL

So, we did not look at usability testing at all, very important very significant area. There is a separate testing for graphical user interface. Again extensive algorithms based testing is available based on finite state machines and so on. We have not looked at any of those and as I have told you 2 or 3 times, by now we have not at all looked at security and reliability testing in detail. So, it is to be noted that there is a fair amount of material available in all these areas and there is active research still going on, but we have not been able to cover them in the course. So, I will end here and I will come back with a module on regression testing.

Thank you.

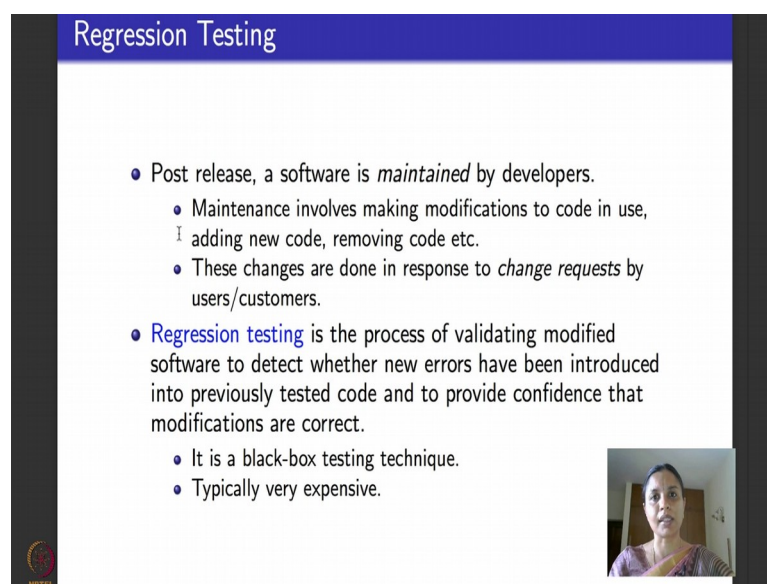
Software testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 58
Regression Testing

Hello again, this is the next module for week 12. I am going to do regression testing today. So, this is again one of the modules that came as request from the takers of the course. So, what is an overview of today's lecture, I will introduce you to what is regression testing, give an overview of it and then one of the biggest problems with regression testing is how to select a set of test cases that is apt for regression testing. So, what is apt depends on what are you regression testing, what is the problem that you are trying to solve and what will be an ideal set of test cases that I can use for regression testing. That problem is the problem of regression test selection.

There are several known techniques for regression test selection because this is one of the important problems. So, we will spend a while understanding an overview. I will give you a survey of 5 different techniques for regression test selection and I will end as we did for the other 2 modules by giving you an idea of some of the tools that I am familiar with to do regression testing.

(Refer Slide Time: 01:21)



Regression Testing

- Post release, a software is *maintained* by developers.
 - Maintenance involves making modifications to code in use, adding new code, removing code etc.
 - These changes are done in response to *change requests* by users/customers.
- **Regression testing** is the process of validating modified software to detect whether new errors have been introduced into previously tested code and to provide confidence that modifications are correct.
 - It is a black-box testing technique.
 - Typically very expensive.

So, what is regression testing? Throughout in this course, we have seen testing as it applies in the software development life cycle. That is while the software is being developed, we do testing, unit testing, integration testing at the software level, at the hardware level system testing acceptance testing and so on. Regression testing comes after all this, this comes after the software is released and is being maintained. Usually it is very rare that a piece of software is released and forgotten about. People release it and also maintain a software. Maintenance of the software involves periodic upgrades that are done to a software, upgrades because some user customer wanted a change in the software, in some functionality of a software or upgrades or changes because there was an error found in the software after release.

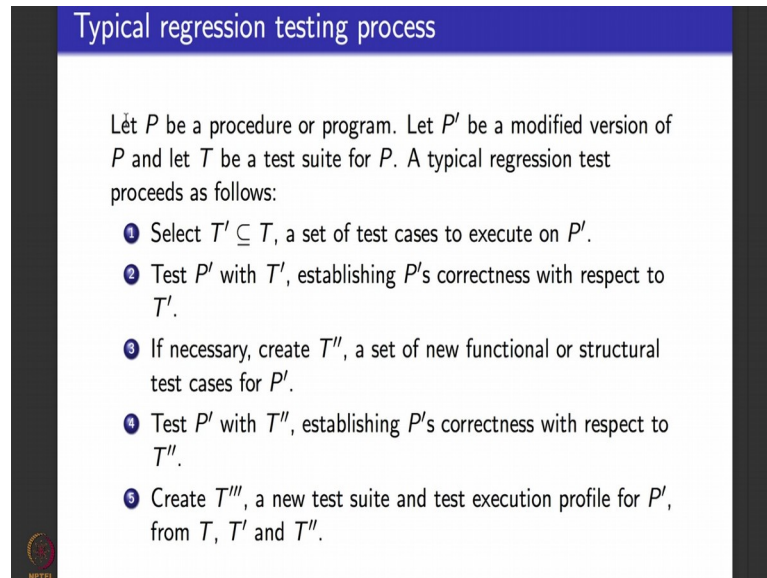
Lot of companies usually develop software that lasts for many years. In fact, in some kind of fields like aerospace and other safety critical system, software is developed to run for many decades, that is 10s of years, 20 years, 30 years, 40 years. Same holds for traditional things like banking software; you know software that was written in 70s and 80s using a programming language like COBOL is still being maintained by several IT companies. How do they do it? They have a team that maintains the software that takes requests for changes, observes errors, writes new code in the language that the software supports and along with writing new code, the team also has to test whether the new code is working fine or not. Such a test that is done by a maintenance team that maintains a software is what is called regression testing.

So, regression testing is a process of validating or testing the modified software. It usually focuses on the piece of software that is modified around the piece of software. A whole software along with the modified component is rarely tested all over again. Obviously, the whole software is put together with a modified software, but the testing focuses on the modifications done and not on repeating the entire set of the tests of the old software was subject to.

So, regression testing is the process of validating or testing the modified software. The goal is to be able to detect whether new errors have been introduced, one, in the modified software and because of the modified software have new errors been introduced in previously tested and released code. Regression testing is also meant to provide confidence that the modifications done are correct. It is typically considered a black box testing technique and is typically very expensive. Why is it very expensive? It is

expensive because people have no idea or clarity about how to select a set of test cases to do regression test.

(Refer Slide Time: 04:52)



Typical regression testing process

Let P be a procedure or program. Let P' be a modified version of P and let T be a test suite for P . A typical regression test proceeds as follows:

- 1 Select $T' \subseteq T$, a set of test cases to execute on P' .
- 2 Test P' with T' , establishing P' 's correctness with respect to T' .
- 3 If necessary, create T'' , a set of new functional or structural test cases for P' .
- 4 Test P' with T'' , establishing P' 's correctness with respect to T'' .
- 5 Create T''' , a new test suite and test execution profile for P' , from T , T' and T'' .

We will spend some time understanding it. So, is it clear please? What regression testing is? A software is being maintained and the process of maintaining a software people make periodic changes to the code. The change that is made to the code is tested to see if it has no errors, if the changed code does not result in any errors in existing software and to gain confidence in the changed modification. Typically what is the process for regression testing. Let us consider P to be a procedure or a program and let P' be a modified version of P the changed version of P and let T be a test suite for P . P was a program that was developed tested and released being maintained why being maintained, it was modified to obtain P' . Now we already have a test suite that was tested for P and documented and stored aside for use during modifications, if any happens.

So, regression tests process proceeds for P' as follows. So, from the set T which is a set of test cases for P , we select the subset T' which is a subset of T which is a set of test cases to execute on the modified version of P . This is a very difficult process. We will spend some time on how to select this T' . Once you have selected T' , what we do is we test the modified P' with the T' establishing P' 's correctness with reference to T' . If there is an error of course, it goes through the natural course. Sometimes in regression testing we also create a new set of functional or structural test cases for the modified version P' . In

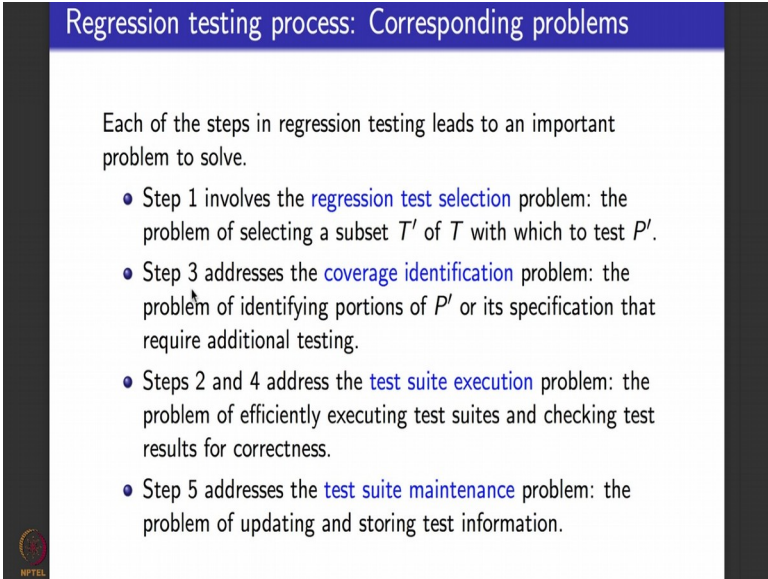
addition to T' we create a new set T'' which is exclusively meant to tests the modified version P' . Then what, once we create it, we test P' with T'' showing that P' is correct with reference to the set of test cases T'' .

Then for now P is no longer exists P' is the one that modified version of P which is going to be integrated into the software and kept now. Later P' might be modified, a portion of P' might be modified. So, you might want to document and record a set of test cases to be used when P' is modified that is the last step. So, a new set of test cases T''' is created from T' and T'' and kept as a record for, as a test suite in a test execution profile to be used for P' .

So, just took succinctly repeat what I said we have a procedure or a program P that is modified to P' , there is a set of available test cases T for P' is tested with. The sub set from P' is tested and P' is also tested with the new set of test cases T'' and P' is tested with T''' which is kept which is derived from T and T'' and T' and kept as a record for the test cases to be used in case P' is modified. So, there are 5 steps that we understood here each step deals with different problem when it comes to regression testing.

So, step one which is this selecting T' which is the sub set of T is called the regression test selection problem. It is a problem of selecting a sub set T' of T with which P' is tested. Step 3, sorry step 3 which is T'' which is a new set of functional test cases is done because sometimes T' may not cover P' .

(Refer Slide Time: 08:03)



Regression testing process: Corresponding problems

Each of the steps in regression testing leads to an important problem to solve.

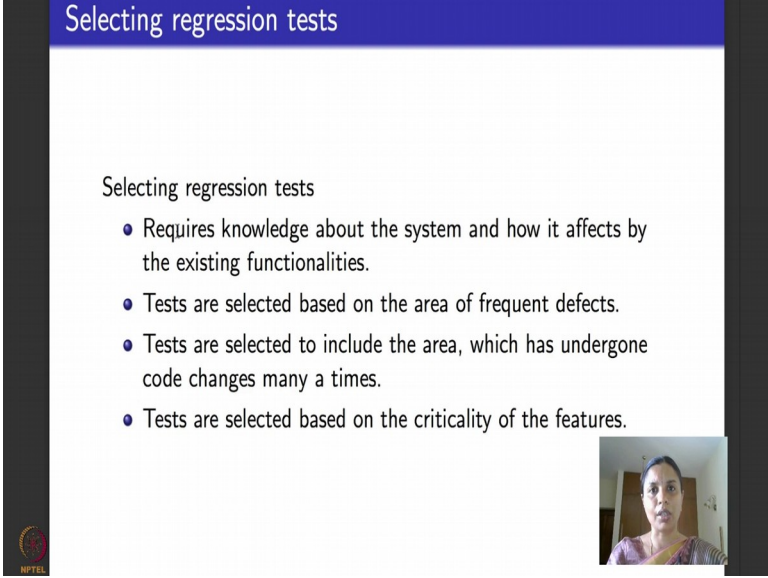
- Step 1 involves the **regression test selection** problem: the problem of selecting a subset T' of T with which to test P' .
- Step 3 addresses the **coverage identification** problem: the problem of identifying portions of P' or its specification that require additional testing.
- Steps 2 and 4 address the **test suite execution** problem: the problem of efficiently executing test suites and checking test results for correctness.
- Step 5 addresses the **test suite maintenance** problem: the problem of updating and storing test information.

So, step 3 addresses the coverage identification problem that is the problem of identifying portions of P' or the specification of P' that require additional testing steps 2 and 4 basically are execution, test case execution. So, it is the set of test suite are selected they are executed.

So, the steps 2 and 4 talk about the test suite execution problem, that is the problem of efficiently executing test suites checking the results for correction. Step 2 and 4 can be fairly largely automated like in other cases of testing because it is just test case execution, does not need much of algorithmic knowledge, but needs extensive tool support. Step 5 is the part where a set of test cases are kept aside to be used for P' if required later. So, step 5 addresses the test suite maintenance problem that is the problem of updating and storing test information. What we will be spending in this lecture is I will tell you some techniques for regression test selection problem which is step 1 and I will also tell you some tools that help you in test suite execution which is regression testing automation. Before we move on just a small point about regression and specifications. Regression test selection is applicable both in cases where specifications have changed and specifications have not changed. When do I do regression testing with specifications have not changed? Maybe I found an error in the software that is released. So, I can develop a batch that rectifies the error and regression testing is done for that.

If this is the case, then it means the specifications have not changed I am doing regression testing because I found an error. So, in the case where the specifications have changed, we have to identify test cases that are obsolete for P' , prior to performing test case selection. So, how, what do you mean by test cases that are obsolete? They are obsolete for P' if the test case specifies an input to P' that is invalid for P' , that means the format of the input, the syntax of the input the type of the input does not match P' . P' cannot even be executed on that input or the test case specifies an invalid output, input output relation for P' . So, once we identify these test cases that are obsolete, our goal is to remove them from T and then do regression test selection on the remaining test cases. So, test selection process, we do not worry about identifying obsolete test cases.

(Refer Slide Time: 10:42)



Selecting regression tests

- Requires knowledge about the system and how it affects by the existing functionalities.
- Tests are selected based on the area of frequent defects.
- Tests are selected to include the area, which has undergone code changes many a times.
- Tests are selected based on the criticality of the features.



The slide features a blue header with the title 'Selecting regression tests'. Below the header, the same title is repeated in a smaller font. A bulleted list contains four points. A small video inset in the bottom right corner shows a woman with dark hair and glasses, wearing a patterned top, speaking. The NPTEL logo is visible in the bottom left corner of the slide.

So, now coming to regression tests selection problem, here is an overview of what happens while selecting regression tests. Regression tests selection requires, obviously, knowledge about the system which could mean domain knowledge and how the modified code affects the existing functionalities without which we cannot effectively tests the code. Tests are selected based on the area of frequent effects. Where do the defects happen? The Pareto principle which is 80% of the defects occur in 20% of the software applies to regression testing also and tests are also selected to include the area which is undergone code changed many times because that is where the problems if any are likely to occur and tests are also selected based on the criticality of features. Let us say you changed because you wanted to cater to a particular feature which is highly critical then you focus on regression testing for that feature, you could do any of these.

(Refer Slide Time: 11:38)

Minimization techniques

- Minimization test selection techniques for regression testing attempt to select minimal sets of test cases from T that yield coverage of modified or affected portions of P .
- The problem, in its generality, is undecidable.
- 0-1 integer programming based techniques are available for procedures that are basically a sequence of statements with single entry and single exit (no branching, looping etc.).



So, we will walk through a set of 5 different techniques for regression test selection. Towards the end of this lecture, I point you to a reference to a survey paper which is slightly old, but still very relevant in terms of its use which talks about all these test selection techniques that we have learnt and papers that I will be referring to, all of them are referred to or cited in the survey. So, the first set of techniques are called minimization techniques. What do they do? They attempt to select minimal set of test cases from the set T that yield the coverage or modified, effected portions of P . Minimal means minimal in number, so the number of test cases is as less as possible.


So, the sub set T' that we choose from T is as minimal as possible usually this problem is un-decidable in its most generic case. I do not know what is the minimal set of test cases, but for these kind of programs or procedure that are basically a one block of statement, a sequence of statements with the single entry and a single exit meaning, there is no branching, no looping, etcetera, there are some solutions based on 0 1 integer programming techniques that do minimization test case selection for regression testing. As I told you the survey paper that I have, I will be referring to towards the end of this lecture will point you to more reference if you are interested in knowing about it.

The next is data flow techniques. Data flow coverage based regression test selection, they select test cases that exercise definitions and use or data interactions that have been effected by the modified portion. Here is one paper slightly old, but again a very good paper.

(Refer Slide Time: 13:03)

Dataflow techniques

- Data-flow coverage based regression test selection techniques select test cases that exercise data interactions that have been affected by modifications.
- For example, the technique of Harrold and Soffa [1988] requires that every definition-use pair that is deleted from P , new in P' , or modified for P' be tested. The technique selects every test case in T that, when executed on P , exercised, deleted or modified definition-use pairs, or executed a statement containing a modified predicate.



It is technique by Harold and Soffa which requires that, what sort of data interaction is tested here every pair of definition and use that is deleted from P or is newly added in P' or is modified for P' , is tested. So, this technique is basically selects a test case T in T such that when it is executed on P , it is exercised, deleted or modified definition use pair or executed on a statement containing a modified predicate. So, it observes attracts values of data definition and uses that have changed from P to P' and develops a set of test cases that basically cater to these changes of definition and uses.

(Refer Slide Time: 14:05)

Safe techniques

- Techniques that are *not safe* can fail to select a test case that would have revealed a fault in the modified program.
- In contrast, when an explicit set of *safety conditions* can be satisfied, safe regression test selection techniques guarantee that the selected subset, T' , contains all test cases in the original test suite T that can reveal faults in P' .
- For example, the technique of Rothermel and Harrold [1997] uses CFG representations of P and P' , and test execution profiles gathered on P , to select every test case in T that, when executed on P , exercised at least one statement that has been deleted from P , or that, when executed on P' , will exercise at least one statement that is new or modified in P' (when P is executed, a statement that does not exist in P cannot be exercised).

The next set of regression test techniques are what are called safe techniques. So, we will understand what is safe technique and what is not safe regression testing. A set of regression test techniques are considered not safe, if they can fail to, they fail to detect the test case that would have revealed a fall fault in the modified test program. Obviously, this is something that we will not desired when we do regression test selection. We want it to be safe in the sense that if there is a fault that is present it is always detected by the set of test cases that I define. So, an explicit it set of safety conditions can be satisfied, a safe regression test selection technique guarantee that the subset T' that I chose contains all the faults that, I mean, contains a set of test cases that can reveal all the faults that are present in P' .



Again, it is quite difficult to, in general, guarantee that I will always be able to define a set of test cases that will detect the faults in P' , but we can try and do our best and this is the paper that does a good attempt. So, here is a paper that basically comes up with the control flow graph representations of P and P' and what it does is that it takes the test execution profiles of T being executed on P and then it says which was at least one statement that was exercised which was actually deleted from P or, the statement when executed on P' , will exercise one statement which is new in P' .

So, basically what it does is that it pulls out set of test cases which make sure that if executed the execution behavior of P' is different from that of P . So, these set of test cases exercise statements that were deleted from P , to get P' or set of statements that were modified in P to get P' and so on, if you do this, then the belief that you are guaranteed to obtain a safe set of regression test techniques which will find a fault in P' if it exists because basically what it tries to do is it tries to cover all the modifications that were done to P to obtain P' .

(Refer Slide Time: 16:15)

Random techniques

- In this case, a randomly selected set of test cases for P are executed on P' .
- This is often useless and done only to show that some regression testing is done.





The next 2 selection techniques have no great technical depth in them, but are the most used in industry ironically. So, what is this one do? This one basically is called a random test selection technique. It randomly chooses a subset the prime of T and tests P' from it. Obviously, because it randomly chosen, there is no bases no coverage and no guarantee that the faults if present in P' will be adopted, but it is a fairly standard practice in industry because there is no time to do regression testing. So, people just randomly select and do a set of test cases. The next is what is called retest all which is basically take all the test cases that were there in the original set T and test P' from it. This is obviously safe because it tests whole thing again.

(Refer Slide Time: 17:00)

Retest-All techniques

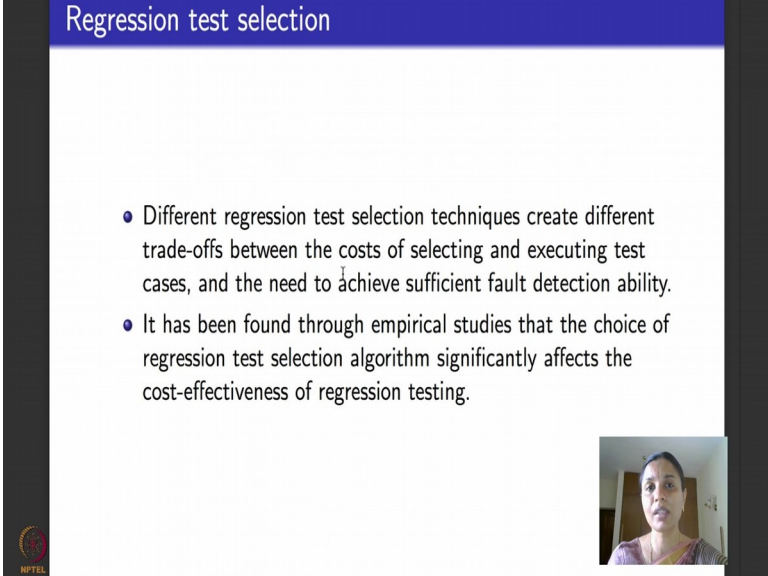
- The retest-all technique simply reuses all existing test cases.
- To test P' , the technique effectively *selects* all test cases in T .



But it may not be efficient because it can be very time consuming and you might have to write some new test cases that are specific to changes done in P' if you just blindly reviews the set of test cases that were using for P , it may or may not be effective for testing P' .

So, we learnt 5 techniques to summarize: minimization which selects minimal set of test cases for regression testing, data flow techniques which select a set of test cases that basically ensure that for every variable that is changed in P' its definition and use attract safe techniques which basically ensure that faults if present in P' are revealed because it executes, selects a set of test cases which either modified or delete a statement in P to obtain P' , random which is basically a random selection of test cases, T' to execute on P' , and retest all it is execute every test case on P that was present in P' . So, these are 5 techniques that are known for test case selection I will point you to a survey paper from which you can learn more about these techniques and also get data to the exact papers that we referred to while giving an overview of these techniques.

(Refer Slide Time: 18:41)



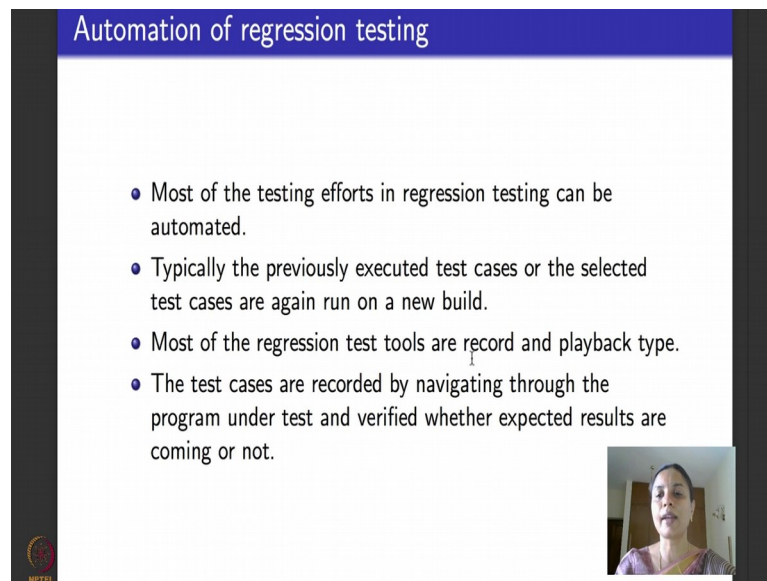
Regression test selection

- Different regression test selection techniques create different trade-offs between the costs of selecting and executing test cases, and the need to achieve sufficient fault detection ability.
- It has been found through empirical studies that the choice of regression test selection algorithm significantly affects the cost-effectiveness of regression testing.

Moving on, now different regression test selection techniques can create trade-offs between the costs of selecting and executing test cases. Obviously, because based on what your priority is you need to select test cases accordingly, it is been found through empirical studies that the choice of the test selection algorithm significantly affects the cost total cost of the regression testing. So, what is, when it comes to automation of

regression testing, like all other testing, this is heavily automated, cannot be manually done on always, especially the execution part of regression testing. So, typically the previously executed test cases or the newly selected test cases are again run on a new build that is created for regression testing. They are basically record and playback type of tools.

(Refer Slide Time: 19:10)



Automation of regression testing

- Most of the testing efforts in regression testing can be automated.
- Typically the previously executed test cases or the selected test cases are again run on a new build.
- Most of the regression test tools are record and playback type.
- The test cases are recorded by navigating through the program under test and verified whether expected results are coming or not.



Which means what happens there? Test cases are recorded by navigating to the program under test and they are verify whether the expected results are coming or not.

So, here are some of the regression testing tools that I have seen. The first one, none of them are open source. All of them are proprietary. The first one is called regression tester that is available from this website. It is pretty decent tool that exclusively focuses on this recording and playback type of execution. Then there is one more TimeShiftX which basically tries to if you typically when you record a test case, you will also time stamp it and record.

(Refer Slide Time: 19:39)

Some regression testing tools

- Regression Tester: Available from <http://www.regressiontester.com/>.
- TimeShiftX: Available from <https://www.vornexinc.com/>.
- Tools from Micro Focus: Available from <https://www.microfocus.com>.





So, this works by telling that you re-executive those state of test cases by shifting the time stamp and which it was recorded. The build is created just by basically manipulating that. The next is a set of tools from micro focus, there are 2-3 tools that do regression testing of some very old code like COBOL programs and all that stuff you can refer to that tool from this website, Micro Focus. Please remember that none of them are open source, I am not aware of very good open source regression testing tools, all of them are proprietary.

(Refer Slide Time: 20:29)

Some references

- Tools: Links already provided, Last access October 2017.
- A nice survey paper on regression test selection:
T. L. Graves, M. J. Harrold, J-M. Kim, A. Porter and G. Rothermel, An Empirical Study of Regression Test Selection Techniques, in ACM Transactions on Software Engineering and Methodology, 30(2), April 2001.



So, to end with here; here is the reference that I told you it is a 2001 survey paper which talks about an empirical study of regression test selection techniques. So, it focuses on

the 5 techniques that we learnt about and tells you how to trade off and how to choose the best test selection problem and some of tools I have already given the link provided please note that these URLs were valid as of this presentation. So, that gives us an overview of regression testing. So, and it pretty much brings us to the end of this course, feel free to ping me in the forum, if you have any doubts for the exam. It is a pleasure interacting with all of you.

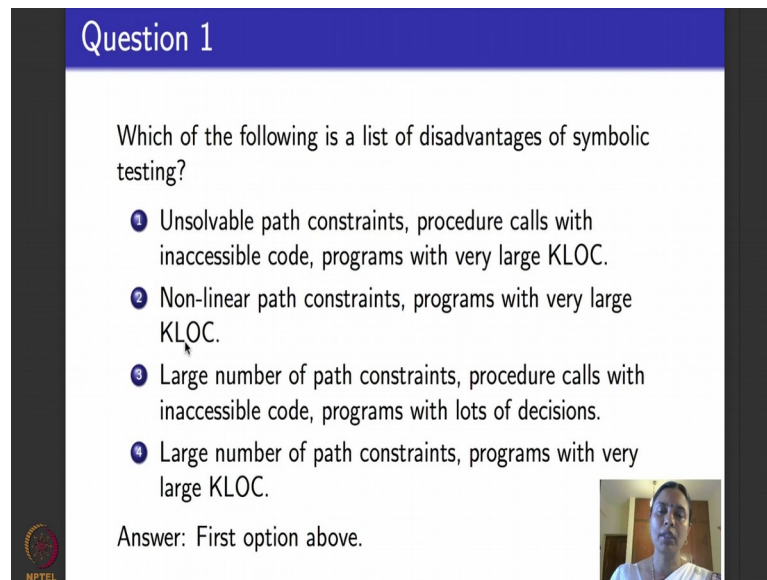
Thank you.

Software testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 59
Assignment: Week 11 Solving

Hello again, this is shall we give on assignment solving. This is the video on assignment solving for week 11, will be uploaded as always after the deadline of the assignment. If you remember week 11 lectures dealt with symbolic and concolic testing and so this assignment is on the questions related to symbolic and concolic testing.

(Refer Slide Time: 00:31)



Question 1

Which of the following is a list of disadvantages of symbolic testing?

- ❶ Unsolvable path constraints, procedure calls with inaccessible code, programs with very large KLOC.
- ❷ Non-linear path constraints, programs with very large KLOC.
- ❸ Large number of path constraints, procedure calls with inaccessible code, programs with lots of decisions.
- ❹ Large number of path constraints, programs with very large KLOC.

Answer: First option above.

NPTEL

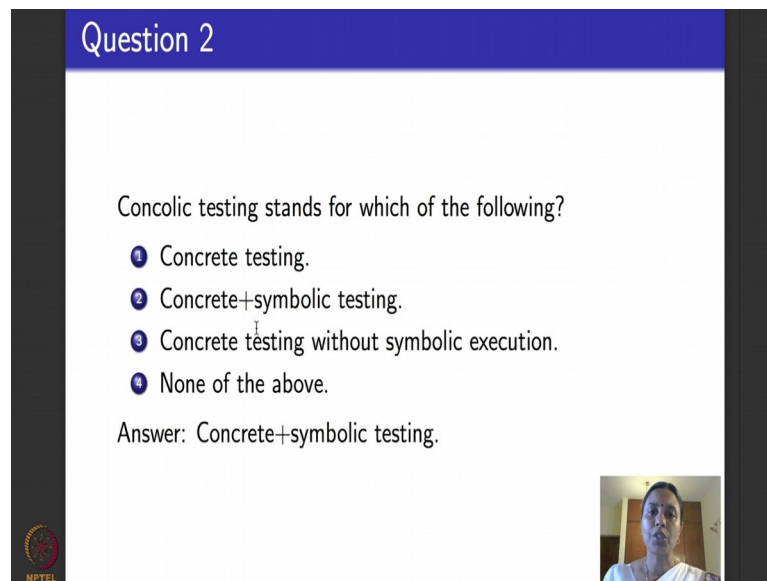
So, as we did for other lectures I will work you through question after question explain the question and also show the answer to you. So, the first question in the video was the following made in the assignment which was the following. Which of the following is a list of disadvantages of symbolic testing? As I told you symbolic testing was introduced in the year 1976, but did not scale up to see the light of the day till the past decade or so, mainly because it had several disadvantages even though it was a novel technique.

So, 4 options were given to you - first one was unsolvable path constraints procedure calls with inaccessible code programs with very large lines of code, second one was non-linear path constraints programs with very large lines of code, third one was large number of path constraints procedure calls with inaccessible code programs with lots of

decisions. Finally, again the same thing but two of the choices that may be repeated. So, while choosing you might be confused to think that second one is a correct answer, but in options like this when there is an overlap on the choices that are available the usual correct thing is to show the complete list of disadvantages. So, from that point of view first option is the correct answer because all three listed here are disadvantages. Unlike option two which lists only two of the disadvantages and options 4 and 3 are not really disadvantages because programs with lots of decisions and the large number of path constraints practically mean the same and that just needs that a longer path constraint, there should not be a disadvantage of symbolic testing.

What is a disadvantage? That is if you get a path constraint that is not solvable cannot be solved by constraints solver, we cannot generate test cases and if you have procedure calls whose code is not accessible then you have a problem because you do not know what additional path constraints should come from the procedure.

(Refer Slide Time: 02:31)



Question 2

Concolic testing stands for which of the following?

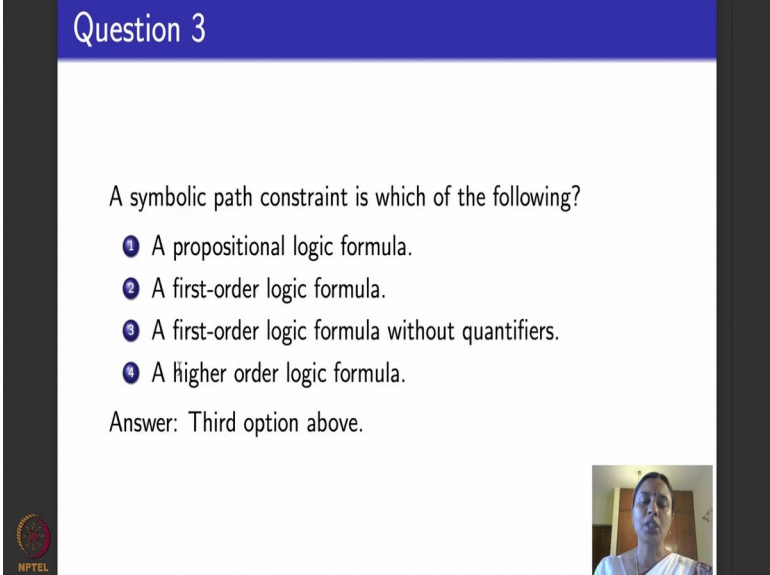
- ① Concrete testing.
- ② Concrete+symbolic testing.
- ③ Concrete testing without symbolic execution.
- ④ None of the above.

Answer: Concrete+symbolic testing.

So, the first one is the correct answer. Second question was what does concolic testing stand for? It was a simple question, four options were given: it just stands for concrete testing, concrete + symbolic testing, concrete testing without symbolic execution or none of the above. So, the correct answer is option 2 because concolic is not a first class English dictionary word, but is derived with a combination of concrete and symbolic.

It tries to do symbolic execution, whenever it encounters one of the disadvantages that we discussed in question 1, it tries to substitute it with concrete values hence the term concrete + symbolic or concolic. So, the correct answer is option number 2.

(Refer Slide Time: 03:21)



Question 3

A symbolic path constraint is which of the following?

- ❶ A propositional logic formula.
- ❷ A first-order logic formula.
- ❸ A first-order logic formula without quantifiers.
- ❹ A higher order logic formula.

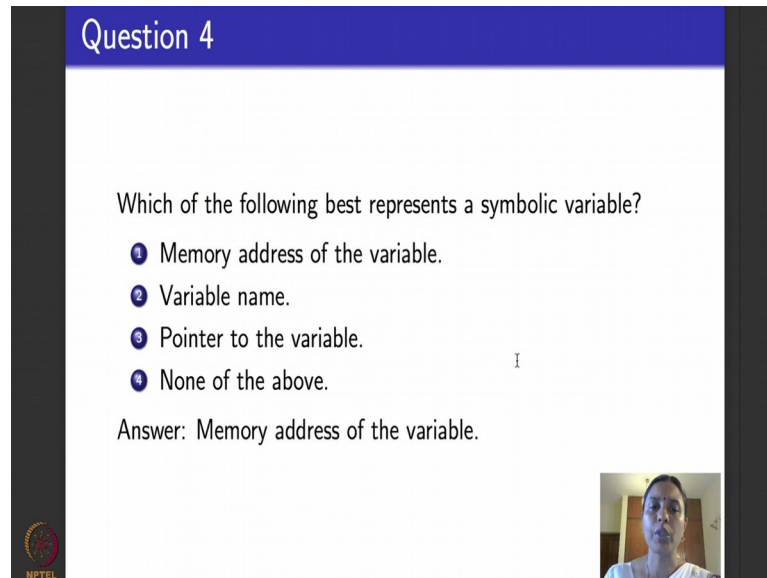
Answer: Third option above.

NPTEL

So, the third question is symbolic path constraint is which of the following? Is it a propositional logic formula, is it an arbitrary first order logic formula, is it a first order logic formula that does not have any quantifiers? Quantifiers are for all and there exists, or is it a higher order logic formula. By higher order, I mean you could quantify over functions and relations also apart from quantifying over variables.

The correct answer is third option. Why is it the third option? It is definitely a first order logic formula, not a propositional logic formula and not a higher order logic formula. Its first order because you have predicates of functions of several variables at the program is working on program has such functions such a first order logic formula. In addition to that there is no scope for getting paths quantifiers in the path constraint you cannot put there exists and for all. So, it is a first order logic formula without quantifiers that is the correct answer, third option.

(Refer Slide Time: 04:22)





Question 4

Which of the following best represents a symbolic variable?

- ① Memory address of the variable.
- ② Variable name.
- ③ Pointer to the variable.
- ④ None of the above.

Answer: Memory address of the variable.



Fourth question says which of the following best represents a symbolic variable. We remember what a symbol variable in symbolic execution was. So, I want to be able to execute the program not with concrete values, but symbolically. Symbolic variables help me to execute a program symbolically and I, taught throughout my lectures, which is gave them some variable names we called them alpha and beta or X and Y.

But if you have to actually go ahead and implement a symbolic execution tool like DART or CREST or CUTE, how would you represent a symbolic variable, that is the question about. So, they were four options again given here, it is a memory address of a variable, it is just a variable name, it is a pointer to a variable, the fourth answer is none of the above. The correct option is the first one: memory address of a variable. So, memory address of a variable can be thought of as a placeholder value that represents the variable symbolically, which can be substituted by a concrete test value that satisfies the constraints that the symbolic value satisfies, the correct answer is memory address of a variable.



(Refer Slide Time: 05:31)

Question 5

In DART, concrete values are substituted instead of symbolic values in which of the following cases?

- 1 Randomly.
- 2 When path constraints are linear.
- 3 When path constraints are non-linear.
- 4 Symbolic values and concrete values are considered alternately.

Answer: Third option above.



Question number 5, in the tool DART that we saw which we discussed from the DART paper, the following question is about that. It asks if the concrete values are substituted instead of symbolic values in which of the following cases. They are substituted randomly, concrete values are substituted when path constraints are linear, concrete values are substituted when path constraints are non-linear or the fourth option, symbolic and concrete values are considered in every alternate run. Which is the correct answer? The correct answer is the third option, which is concrete values are substituted by the DART tool when path constraints are non-linear. Why is this a correct option? If you remember the time in which a paper like DART tool was developed was in 2004-2005. At that time when it comes to constraint solving, there were no good constraint solvers that could handle non-linear path constraints.



As I told you during the lectures in that week, the generic problem is an undesirable problem. So, in DART they took a decision that whenever path constraints are non-linear we could substitute it with concrete values and go ahead and drop symbolic execution in such cases. So, the correct answer is non-linear path constraints or option 3.

(Refer Slide Time: 06:51)

Questions 6 (i)–(v)

For the next five questions, we consider the following program segment.

```
int x, y;  
1: if (x > y) {  
2:   x = x + y;  
3:   y = x - y;  
4:   x = x - y;  
5:   if (x - y > 0)  
6:     assert(false);  
}
```



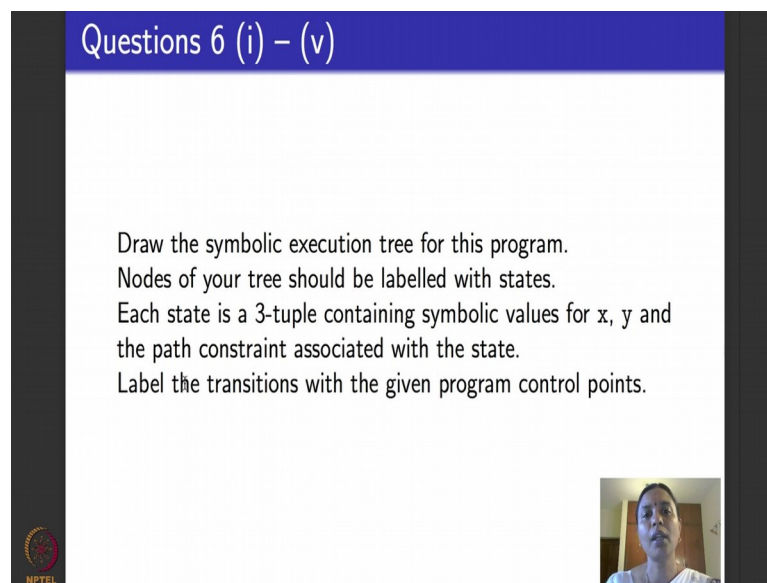
So, for the remaining five questions in the assignment for last week I had given you a piece of code and asked you to draw the symbolic execution tree for that piece of code and there were questions about symbolically executing that code. Like we saw throughout this course this is not a complete piece of code, it is just the fragment of the code. We will first go through the code that was given in the lecture and see what it is doing.

So, it took two integer variables x and y , there were as an a statement which said x is $> y$ then you do these 3 statements. I will come back in a minute and tell you what these 3 statements are. And then it says if $x - y$ is $>$ zero then you assert false. So, there are two if statements, one nested inside the other and then if we reach line number 6, it means that you have reached an error, assert false means the at time, the program has an error, interpreted that way; that you are asserting something that is wrong if you reach this line number 6 which means the program has an error. So, what do these lines 2, 3 and 4 the program statements do? So, what it says is, if x is $> y$, let us say x is 5, y is 2 just as an example, the first one does x is $x + y$, it adds it. So, make x is 7 as per our values and then it does y is $x - y$.

So, what is x now after the previous statement? It is $x + y$. So, you substitute it here you will get $x + y - y$, the $+ y$ and $- y$ will get canceled and y will take the value of x . The last, the fourth line says x is $x - y$. So, what does x is $x - y$ mean? This x here on the right

hand side is actually $x + y$ and the y is the actually x . So, what it will do is, x is $x + y - x$, the x and $-x$ will again get canceled, x will become y , so basically lines 2, 3 and 4 swap the values of x and y . I hope that is clear. It is not the most obvious way of swapping, the most obvious way of swapping that we know is to have a temporary variable, assign one value to the temporary and then do this. This just does swapping in a slightly different way, that does not matter to us, but it basically swaps x and y , that is all it does. And say shifts swapped values of x and y are such that $x - y$ is > 0 then for some reason I have hidden error.

(Refer Slide Time: 09:25)



Questions 6 (i) – (v)

Draw the symbolic execution tree for this program.
Nodes of your tree should be labelled with states.
Each state is a 3-tuple containing symbolic values for x , y and the path constraint associated with the state.
Label the transitions with the given program control points.

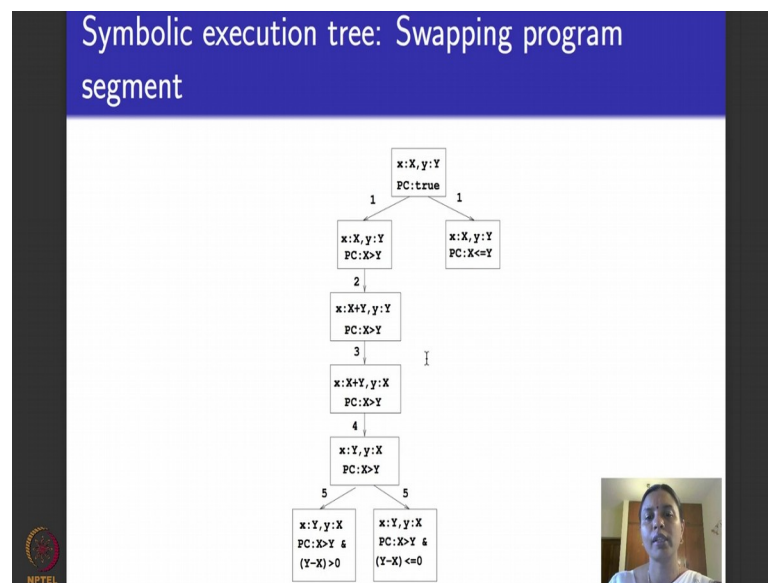
NPTEL

A small video inset of a person speaking is visible in the bottom right corner of the slide.

So, this was the example of program fragment that was given and then you were asked to draw the symbolic execution tree for this program. Based on the symbolic execution tree there were 5 more questions asked in the assignment. So, here were some guidelines on drawing the symbolic execution tree. Symbolic execution tree is like any other graph, it will have vertices and edges. So, the vertices that the nodes have here trees should be labelled with the states. What do we mean by states? If you remember what is the state in symbolic execution, it is the values of the variables. So, here they were two variables x and y . So, each node of vertex should be labelled with those values and then here it explains what the state is, it is a value, it is a 3 tuple consisting of symbolic values for x and y . Along with, in symbolic execution we keep a path constraint which is what are all the conditions to be satisfied on the variables for me to reach a particular statement so far.

So, I keep these things in a node and the transitions of the symbolic execution tree. The guideline was to label the transition with the given program control points. What do we mean by control points? They are just the statement numbers in the program. So, using these guidelines and from the lectures, here is how you would have drawn a symbolic execution tree. Please do not worry, if exactly you did not draw it is this way, but you would should have got something with 8 nodes that has this kind of a branching structure that would have been a correct symbolic execution tree.

(Refer Slide Time: 10:37)



So, what is the guideline? Let us go back and look at the nodes. The vertices of your tree should be labelled with states, state is a 3 tuple consisting of symbolic values for x and y and the path constraint.

So, if you see that is what I have done here. They were two variables, small x and small y, I have given their symbolic values with the names X and Y. So, here is the first state x is the variable small x is assigned the symbolic value X. The variable y small y is assigned the symbolic value Y and we get to begin execution in the program. So as always, PC stands for path constraint, to start with is true. So, if you go back and look at the code this is where I am. Very first statement in the program is a decision statement, $x > y$. Then you do this, if x is $\leq y$ then you exit. So, there is a branch coming out of this which is labeled by the one number 1, which is the first if statement. This is when x is $>$

y the path constraint is added to this true and this will just be this. So, I have retained it as $x > y$. This is negation of $x > y$ which is $x \leq y$.

So, in the beginning first statement can test positive in which case I reach the left branch, first if statement can test negative in which case I reached the right branch. If I reach the right branch the execution stops there, there is nothing else in the program to execute. So, we go back and see what happens in the left branch when the if statement is true. So, I enter the if statement if you go back and look at the program they were 3 statements label with numbers 2, 3 and 4 which we discussed they basically swapped the values of x and y. So, I am capturing them in the symbolic statement, symbolic execution tree. So, after statement number 2 is executed x becomes x + y, y becomes y that that is true right because x becomes x + y statement 2 does not change the value of y.

So, y stays as Y, that is what is captured here. Path constraint is $x > y$ because I am within the first if statement. After the second statement, statement number 3 changes the value of y, so x continues to stay as $X + Y$, Y becomes X. So, this is where y gets the value of x. Path constraint is the same after the third statement. You will realize now here, see x has taken the value for y and y has taken the value of x. Before these three statements were executed you go back and look up in the node, here x was taking the value x, y was taking the value y. After executing these three statements I have swapped the values of x and y, path constraint continues to remain the same. So, where are we in the program? We finished executing line number 4, statement number 4.

The statement number 5 is an if statement it says if x is greater, $x - y > 0$ then you assert false otherwise you exit. So, that is the symbolic execution here. So, statement 5 is an if statement. So, there is a branching in the execution tree. So, path constraint is whatever you had earlier. We had $x > y$ earlier. If you see throughout I am AND-ing that with the new path constraint which is $y - x$. Why I have a changed to $y - x$, when $x - y$ was given here? Please remember after the three statements, 2, 3 and 4, the values of x and y are swapped. So, x becomes Y, y becomes X, X and Y with a symbolic execution values for small x and small y. So, path constraint is $x > y$, whatever they earlier and $y - x > 0$. I have passed the second if statement here. The path constraint is $x > y$ and $y - x \leq 0$. So, if I pass this, I reach an error. I have not depicted that in my symbolic execution tree. If you want to you put another node here which says that you go to line

number 6 and say assert false. Because it is an error statement and asserting false means abortion of execution program execution, I have not depicted that.

So, this is how the symbolic execution tree will look like. So, the questions, the remaining 5 questions were around this execution tree and what happens during symbolic execution.

(Refer Slide Time: 15:22)

Question 6 (i)

How many nodes are there in the symbolic execution tree?

- 1 5 nodes
- 2 6 nodes
- 3 7 nodes
- 4 8 nodes

Answer: 8 nodes.

So, first question is very easy if you are drawn the symbolic execution tree correctly. The question just asks you how many nodes of vertices are there in the symbolic execution tree. As you could see here if you count there are 3 here, $3 + 3 - 6 + 2 = 8$ nodes in the symbolic execution tree. So, the correct option was fourth option.



(Refer Slide Time: 15:42)

Question 6(ii)

What is the path constraint for the program execution that goes through the statements 2, 3 and 4?

- ① $x > y$.
- ② $x < y$.
- ③ $x \geq y$.
- ④ True.

Answer: First option above.



The next question says what is the path constraint for program execution that goes through the statements 2, 3 and 4. So, let us go back statements 2, 3 and 4. What was the path constraint when do I reach statements 2, 3 and 4? If the if statement $x > y$ turns out be true. So, the path constraint for all the three executions labeled in the symbolic execution tree is $x > y$.

So, there were 4 options given here: $x > y$, $x < y$, $x \geq y$ or true. The correct answer is $x > y$, it is the first option.



(Refer Slide Time: 16:15)

Question 6(iii)

What is the path constraint when the program segment begins execution at statement 1?

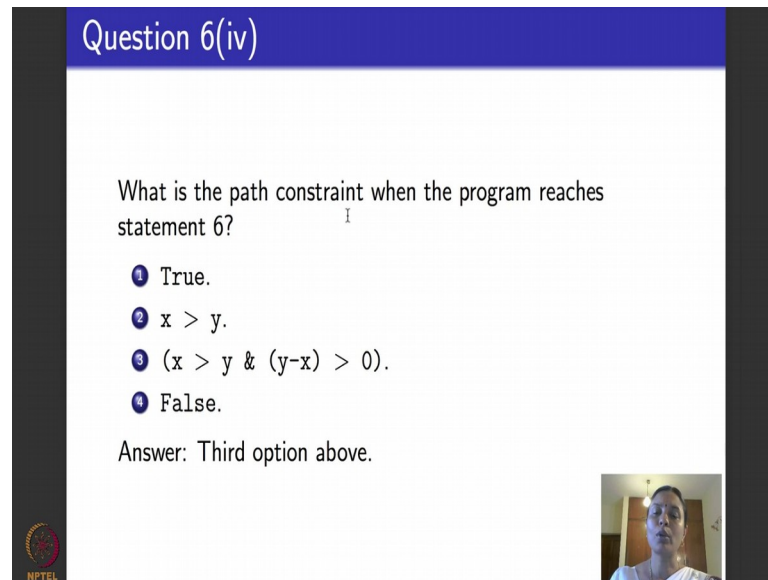
- ① True.
- ② False.

Answer: True.



So, the third question was, what was the path constraint when the program begins execution at statement number 1? Here the program segment if you see, there was nothing to begin with, no constraint. So, the path execution when I begin execution is just the constant true, that is what is given here. So, the answer is true.

(Refer Slide Time: 16:33)



Question 6(iv)

What is the path constraint when the program reaches statement 6?

- 1 True.
- 2 $x > y$.
- 3 $(x > y \ \& \ (y-x) > 0)$.
- 4 False.

Answer: Third option above.

The next question says what is the path constraint when the program reaches statement 6. So, four options were given true $x > y$, $x > y$ and $y - x$ is > 0 and false. The correct answer, if you go back and look at the execution tree this, after this, is when statement number 6 is reached. So, the path constraint is $x > y$ and $y - x$ is > 0 , that is what is given here. So, the correct answer that you should have chosen was the third option.

This is before the second if statement this event of write and the beginning path constraint was not false we were only asserting false. Last question was very simple, it asked will the statement number 6 in the program ever be reachable. The answer is yes. It will be it is simple, you do not even have to draw the symbolic execution tree, we just go back and see this program. To reach statement number 6 I have to make this if statement true and I have to make this if statement true. I can obviously give values x and y such that this is true. So, statement number 6 will obviously, be reached by the program. So, the correct answer to the last question and the assignment was yes which is the first option.

So, I hope this video helped to you to solve the assignments. Please feel free to ping me and the forum if you have any further doubts about this particular assignment.

Thank you.

Software testing
Prof. Meenakshi D'Souza
Department of Computer Science and Engineering
International Institute of Information Technology, Bangalore

Lecture – 60
Software Testing: Summary at the end of the course

Hello everyone, this is the absolute last lecture of the software testing course. What I thought I will do in this lecture is to give you an end of the course summary. We will revisit all that we did in the course, find out what is not done and maybe look at some of the tools that will help us to do software testing. If you remember and if you listen to all the lectures of the course towards the end of 9th week, I had done a similar such summary because at that time, we had looked at majority test case design and criteria and I thought it was apt at that time to give a summary through three-fourths of the course.



This lecture is essentially a repeat of that summary along with a few additional details that we have covered towards the end of the course. One of the things that I consciously took a decision of and did not do in this course is to teach you tools. So, what I thought I will do is towards the end of the course, I will give you an overview of about approximately ten tools that are widely used or believed to be widely used, to the extent I could gather information, across different firms and institutions that to test their software.

So, this course will essentially, this lecture will essentially be an overview recap of all that we have done in the course similar to the one that we did in week 9, along with an overview of software testing tools that are currently been widely used.

(Refer Slide Time: 01:46)

Course overview

- This course will cover algorithms and techniques for test case design based on models of software artifacts.
- Test cases will be designed based on graphs, logical predicates, input domains and on the syntax of programming languages.
- The test cases designed will be applicable for both black-box and white-box testing, covering a broad range of languages, platforms and applications.





So, again I said it for week 9, I am starting with what was committed in the course that we began in the beginning. So, this course we said will cover algorithms and techniques for test case design by basing, by modelling them, the software artifacts through 4 kinds of models: graphs, logical expressions, input sets and models based on grammars.

So, then we said these test cases could be applied to test code, white box testing or could be applied to test based on requirements, pre-conditions, post-conditions and invariance which would be considered as black box testing.

(Refer Slide Time: 02:30)

Course Contents

- Introduction, software testing process levels, testing terminology
- Techniques and algorithms for test case design:
- Graphs based testing
 - Structural coverage criteria, data flow coverage criteria
 - Graph coverage for source code, design elements and specifications
- Logic based testing
 - Predicates and clauses, coverage criteria based on logic expressions
 - Specification-based logic coverage
 - Logic coverage for finite state machines



So, the contents of the course, as I recap from what was posted for you all to see and it will be in the website, reads as follows. We began the first week by introducing software testing; what are the process levels, the basic terminologies clarified a lot on them. Then the bulk of the course from about the second week till about the ninth week, dealt with techniques and algorithms for basically designing test cases. So, we considered software artifacts, which included requirements, design, source code, elements of design like pre-conditions, post-conditions, special models like finite state machines and so on and modeled them first using graphs.



We learnt some graph algorithms for that we looked at structural coverage criteria purely based on graphs, then we looked at augmenting the structural coverage criteria with data flow which is the basis of a course and program analysis also which talked about definitions and uses of variables. Then we applied the structural coverage criteria and the data flow coverage criteria to graph models derived from source code, from design elements, finite state machines and from specifications. After this coverage on graphs, we started with logic I gave you a very very brief introduction to logic. We saw what predicates were clauses were, what is propositional logic, how difficult are the algorithms, checking satisfiability of propositional logic formulae and predicate logic formulae.

And then we looked at coverage criteria based on logical expressions. We looked at simple coverage criteria like predicate coverage, clause coverage, then we also looked at interesting criteria like active clause coverage. Three versions of it, inactive clause coverage; two versions of it. These are widely used for testing safety critical systems. Then we took this logical coverage criteria. First saw how to apply it to source code. If you remember, we saw 2 examples; example of a thermostat; an example that determines the type of a triangle. Then we looked at how to apply logical coverage criteria for design elements where we looked at pre-conditions and how to apply logical coverage criteria for state machines. Moving on, we looked at black box testing. I introduced you to functional testing as you would find in many popular text books.

(Refer Slide Time: 04:56)

Course contents, contd.

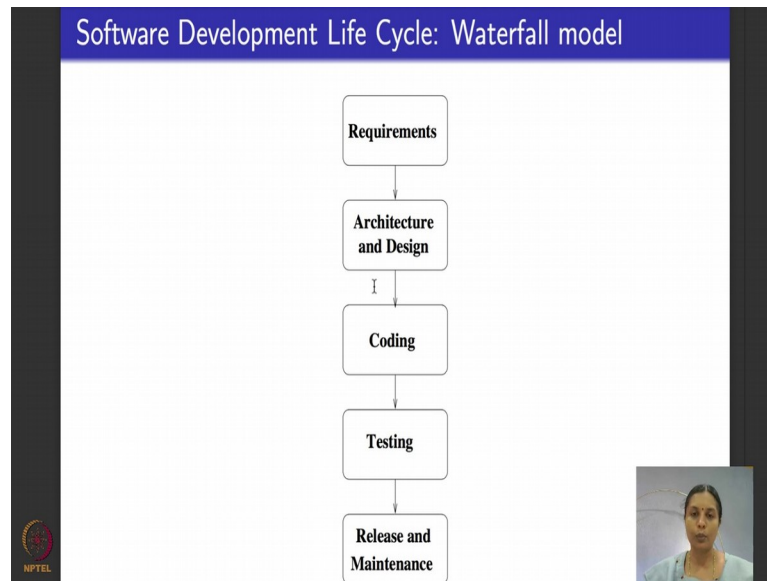
- Input space partitioning: Input domain modeling, combination strategies criteria
- Syntax based testing: Coverage criteria based on syntax, mutation testing
- Test case design (as learnt above) applied to
 - Testing OO-applications
 - Testing web applications
 - Testing embedded software
 - Testing GUI
- Symbolic testing and concolic testing



We learnt equivalence partitioning, boundary value analysis, decision tables, random testing and so on. Then we looked at a generic technique that partitions the input domain and derives test cases by considering combinations of these partitions. We saw how to model the input domain, how to partition the input domain and we saw about six different criteria to model combinations of input domains and write test cases for them. If you remember, we said all combinations coverage, each choice coverage, pair wise coverage, T-wise coverage and for functional testing based input domain modeling, we looked at based choice coverage and multiple base choice coverage. So, that was a break from logic and graph based testing. We did black box input space partitioning based testing. From there on, we moved on and looked at mutation testing which involved testing based on grammars.

I first introduced you to grammar as it occurred in programming languages. To be able to do that, we saw regular expressions, context free grammars. Then we saw coverage criteria based on syntax mutation testing coverage criteria and then what it is say in the course, we said we are going to learn it to tests object oriented applications, web applications, embedded software, GUI and then we said we will tests symbolic testing and concolic testing.

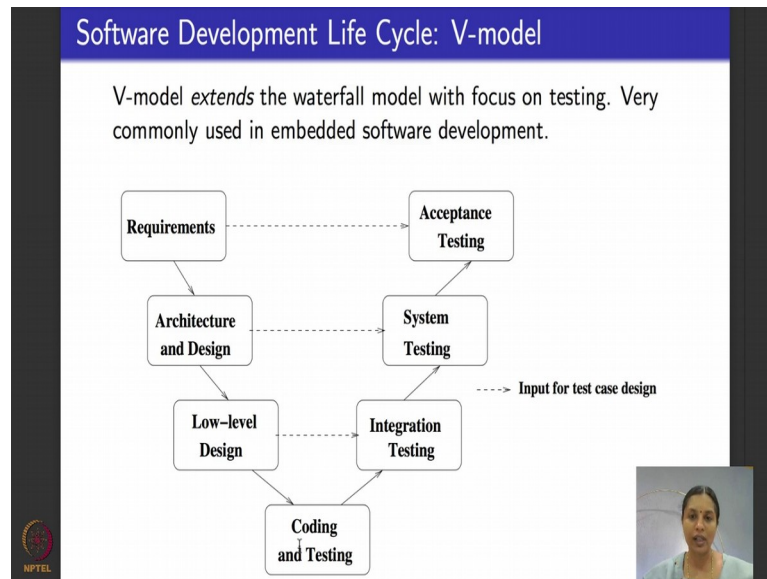
(Refer Slide Time: 06:35)



So, if you remember, this is how a software development lifecycle works. It is a classical waterfall model. We begin with writing requirements high level system level requirements drill down to hardware software and specific requirements. Follow it up with architecture and design, follow it up with coding and unit testing and then integration and system level testing and then release and maintenance.

Testing applies throughout this cycle and how does testing apply throughout this cycle? We bend the waterfall model to be able to get what is called the V model. Waterfall model is this part, a part of it is depicted on the left hand side of the V model as they say, this is just a copy of what is given here; requirements, architecture, low level design, I have refined it a bit.

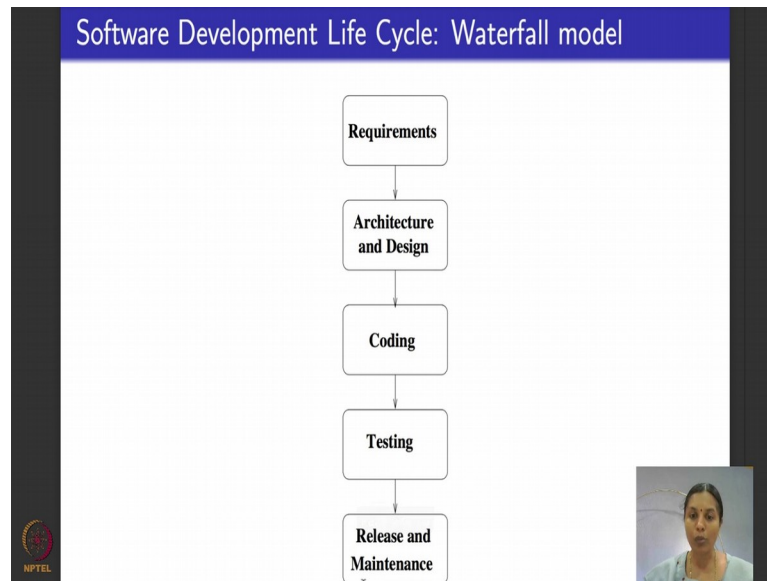
(Refer Slide Time: 07:06)



Because I want to be able to link it to integration testing, coding and testing and what I did in the last phase here was to expand out. Testing comes along with coding which is unit testing followed by integration testing for which inputs or design for test cases come from the low level design because that tells you how a software is broken up into components of modules, followed by system testing. For functional to system testing inputs come from the design document. System testing involves putting the entire thing together and testing if the overall software integrated with the system needs this functionality.

As we saw in the last lecture, in this week's lecture on non-functional testing system testing also includes non-functional testing. Non-functional testing tests for various quality attributes like reliability, security, usability, performance, interoperability and we saw a whole set of nonfunctional testing attributes. Where does the majority of inputs for designing such test cases come from? They come from the architecture document which gives a static picture of the various components of the software, how they interact which is the processor in which it is going to run on what is the platform care features and so on. After all this is done finally, users, end users of the software do acceptance testing to check if the software and the system meets its desired requirements or not.

(Refer Slide Time: 08:56)



In fact, in this week, we saw some testing after the acceptance testing which is basically going back to the waterfall model, deals with this phase of software development which is release and maintenance.

I gave you an overview of regression testing which people do when they maintain software to cater to patches, upgrades or change requests that happen in the software.

(Refer Slide Time: 09:16)

Types of Testing

There are different types/levels of testing, based on the phase of software development lifecycle that they are applied to:

- **Unit Testing:** Done by developer during coding.
- **Integration Testing:** Various components are put together and testing. Components could be software components or software and hardware components.
- **System Testing:** Done with full system implementation and platform in which the system will be running.
- **Acceptance Testing:** Done by end customer to ensure that the delivered products meets the committed requirements.
- A related term is **beta testing** which is done in a so-called beta version of the software, by end users, after release.

The NPTEL logo is visible in the bottom left corner.

So, these are the various sets of terminologies that we have seen. Each one tries to attempt testing and classify it into categories based on a particular feature or a phase that the testing is involved in. The 5 types of testing that we saw are unit testing which is

done here, by the developer, especially methodologies developers are expected to do unit testing, you cannot rely and hope that there will be a separate tester who would do it for you, followed by integration testing, here components are put together. Components could be software software components or software hardware components. We specifically saw integration testing related the software software components in this course and not integrating it with hardware. Then system testing is done with the full system in place, software running on the desired hardware platform.

Finally acceptance testing is done by customers and a related term is beta testing which is when the software is released, but with known bugs not 100 percent guarantee that it will cover and people do beta testing as a version of acceptance testing.

(Refer Slide Time: 10:27)

Testing Methods

There are two broad methods of testing:

- **Black-box testing:** A method of testing that examines the functionalities of software/system without looking into its internal design or code.
- **White-box testing:** A method of testing that tests the internal structure of the design or code of a software.

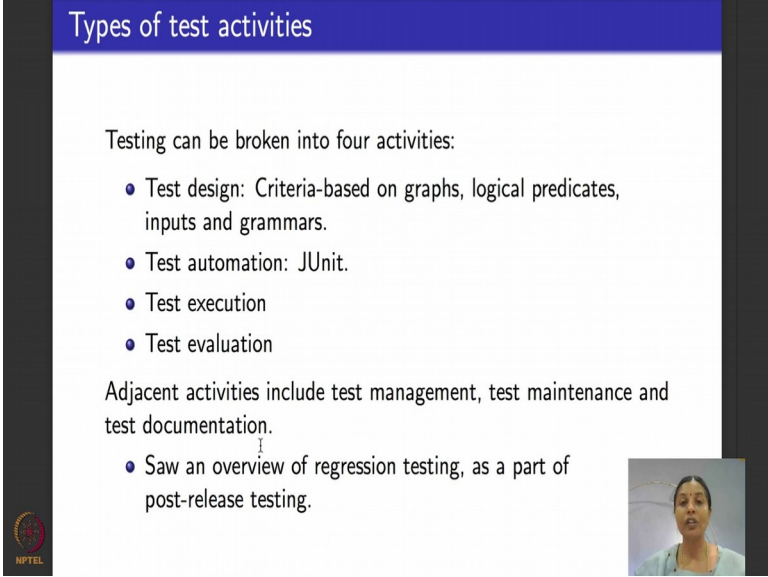
Black-box testing	White-box testing
Unit, integration, system and acceptance testing	Unit, integration and system testing
Usability, performance, beta and stress testing	

So, this spans the V model as we saw it. Another classification of testing are what is called methods in testing: black box and white box. We saw these in detail. Black box testing considers the software or the system as a black box. Only the executable is given, we do not look at its internal details. White box exploits the structure of the considered software to be able to design test cases.

Black box is widely practiced across several different testing phases and stages. White box is typically restricted to unit testing, integration testing at software software level and at the system level. A lot of the course, we did see a mixture of white box and black box testing as it applies to the first row, here we did not spend too much time for black

box testing as it applies to the second row here. The last week, in this lecture in this week, I gave you an overview of some of these techniques.

(Refer Slide Time: 11:23)



The slide is titled "Types of test activities" in a blue header. The main content area is white and contains the following text:

Testing can be broken into four activities:

- Test design: Criteria-based on graphs, logical predicates, inputs and grammars.
- Test automation: JUnit.
- Test execution
- Test evaluation

Adjacent activities include test management, test maintenance and test documentation.

- Saw an overview of regression testing, as a part of post-release testing.

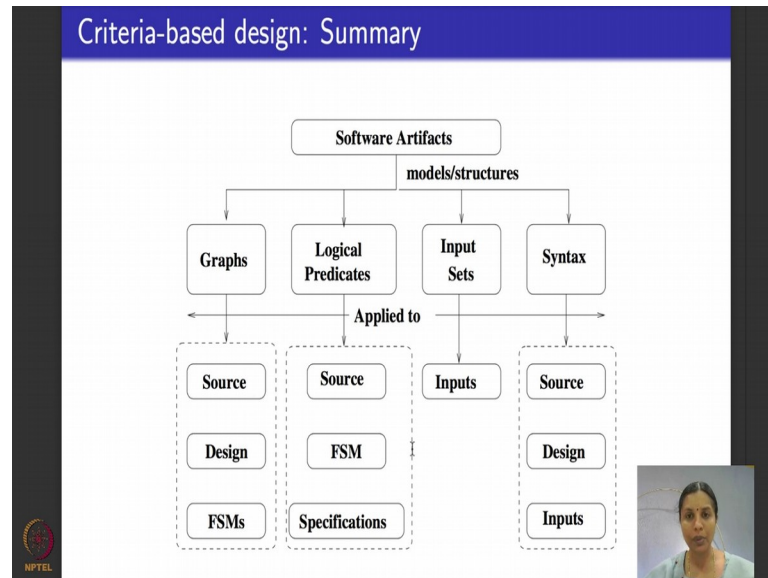
In the bottom right corner of the slide, there is a small video inset showing a woman speaking. The NPTEL logo is visible in the bottom left corner of the slide frame.

Another related thing is what are the activities or the processes of testing. The first thing is after you have your testing goals is to be able to design your test cases; that is what we did most of the course. We did an algorithmic style of designing the test cases and then once you design a test case you pick your tool of choice for execution. For this course, I had introduced you all to the tool JUnit because it was popularly used for Java and the book that I followed for majority of the course also uses JUnit test cases. So, that is why I had introduced that tool to you, I hope you were able to try out some examples with JUnit. Once you have JUnit and you figure out how to automate the test script on JUnit, the next job is to execute it and evaluate the test case to see if the test case is actually revealed an error in the program or not.

Typically lot of other activities happen along with testing: test management, test maintenance. documenting the test cases. Typically if these activities have become very important after a software is released when it is being maintained here because that is when you do regression testing as we saw in this week. And the sources, documents that are needed for regression testing are the documents that you keep with the goal of test maintenance which is basically reusing with set of test cases for regression testing, and documentation which talks about documenting a test cases which will help you to select

which test cases to reuse for regression testing. Then this is a summary of the criteria based design that we saw, I just repeat it to you.

(Refer Slide Time: 12:55)



So, I will be brief here. Software artifacts, could be code source, code design, preconditions, post conditions assertions, specifications, models like finite state machines, activity diagrams and so on.



I take them based on what I want to test, I can model it as a graph model, model it as a logical predicate consider only the inputs that drove the software artifact or work with the syntax of the artifact. Once I model the software artifact using one of these four, I can use it to test source code design finite state machines, I can again use it to test source code finite state machine and specifications, but where I consider input space partitioning its only black box.

(Refer Slide Time: 13:55)

Classical testing: As seen along the way

We saw several classical testing terminologies and techniques along the way:

- Source code:
 - Cyclomatic complexity, independent paths and basis path testing, decision-to-decision paths.
 - Statement coverage, branch coverage, loop coverage.
- Design integration testing:
 - Interfaces and their types.
 - Stubs and drivers.
 - Incremental, top-down, bottom-up, bigbang and sandwich approaches.
- Black-box testing techniques:
 - Decision tables, equivalence partitioning, boundary value analysis.



So, I can use it only to test a software with reference with to its black box testing. This was the main bulk of the course as I taught you from the book by Amman and Offutt. Along with this, we also saw several classical software testing terminologies, typically when you pick up other books in software testing, when it comes to testing source code, you would hear of terminologies like cyclomatic complexity, independent paths, basis paths testing, D to D paths, statement coverage, branch coverage, loop coverage, path coverage and so on.



We did look at all these terms because these are classical standard terms and software testing. Now I hope it is clear how these terms relate to the coverage criteria that we saw in the course. Then when it comes to design integration testing, I told you about top down integration, bottom up integration, the various kinds of interfaces, message passing interface, client server interface, shared variable communication, then we saw what test stubs and test drivers were, how to create stubs, how to test using stubs and drivers and typically do top down or bottom up integration, which are the 2 most popular integration testing techniques. In black box testing techniques, I did a module on functional testing where you learnt about equivalence partitioning, boundary value analysis and decision tables.

(Refer Slide Time: 15:09)

Formal languages

Presented an overview of

- Propositional logic, basics of predicate logic.
- Finite state automata.
- Regular expressions.
- Context-free grammars.





In the process, we also learned a bit of discrete maths and formal languages. I taught you graphs, graph algorithms DFS-BFS; topological sort, strongly connected components. Then we looked at logic specifically propositional logic and basics of predicate logic; how difficult is the satisfiability problem for propositional logic its NP-complete for predicate logic is un-decidable. So, we work with a lot of heuristics bundled together and sat solvers to be able to solve the logical formulae. We also saw a detailed introduction to finite state machines, to regular expressions, to context free grammars, as they were used to build grammars or the syntax of programming languages.

(Refer Slide Time: 15:50)

Programs, design models and FSMs

- Several examples of source code.
- Examples of sequencing constraints, pre-conditions and post-conditions.
- Examples of finite state machines for specifications.



So, we saw several examples of source code. If you try to recap the examples that we saw throughout the program, we saw triangle types, then we saw thermostat, several different examples. Then we saw examples of sequencing constraints, preconditions, post conditions, if you remember calendar method, queue example, file open, file close. Then we saw examples of finite state machines for specification, subway, microwave oven and so on.

(Refer Slide Time: 16:27)

Testing different kinds of software

The following topics were discussed:

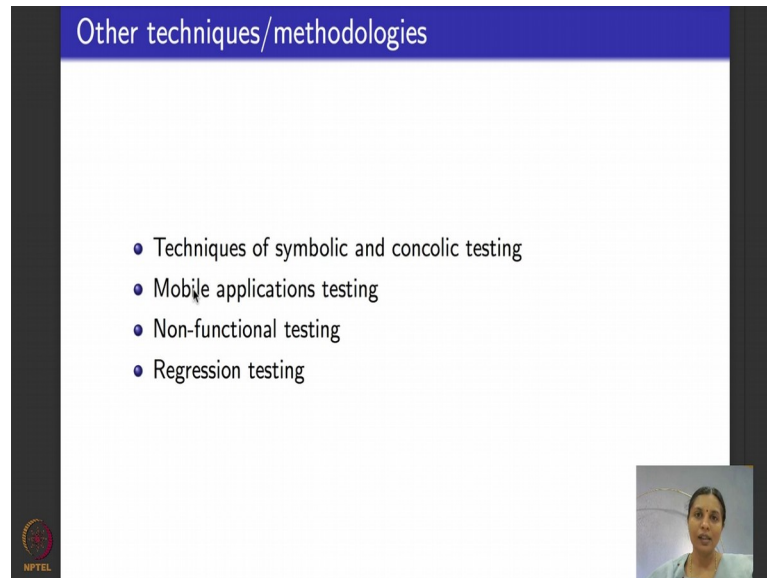
- Testing of web applications
- Testing of object-oriented software

NPTEL

So, we also after week 9, this was what I did newly after week 9, a lot of what I told you now was a repeat of the overview that I had given at the end of week 9. After week 9, we mainly concentrated on the following topics. We saw how to test web applications, static hypertext websites, dynamic hypertext websites from the client side and from the server side. I had taught you one technique each for server side testing and for client side testing: bypass testing, user session data based testing and testing based on graph models which were derived from the code corresponding to server as atomic sessions. And then we also looked at testing of object oriented software because typically object oriented software have lot of features like inheritance, polymorphism. We did a sort of a crash introduction to all these object oriented features. We saw how these features can lead to lesser static determinism and you do not know which version of the variable is being called and how to determine the state.

We saw the models of yo-yo graphs, we saw various object oriented faults in detail and towards the end in this week, I defined object oriented integration testing coverage criteria for you all. Last week, week 11, we saw symbolic and concolic testing techniques which was a distinctive departure from the coverage criteria based techniques that we have seen until now.

(Refer Slide Time: 17:41)





In concolic testing, we particularly saw the dart tools algorithm then as requested by the users of this course I gave an overview of these topics.

Mobile applications testing; testing non functional things with specific focus on performance testing and regression testing. While I did this, I also gave you references to some tools, open source and proprietary, that you could have access to or use it to try it out and what we did not do in the course. What else is there in software testing that we did not do over the course. First of all I hope you do believe that we have covered a lot in the course and I hope it was useful learning for you.

(Refer Slide Time: 18:22)

What we didn't do?

- Non-functional testing, in detail.
 - Security testing, software reliability analysis, usability testing.
- Testing of GUI.
- Testing of embedded software.
- Tools, apart from JUnit.
- Acceptance testing.



But, like in any other course in any other topic we have to leave out a lot of things. So, what we did do is non-functional testing in detail. I did one module of non-functional testing, we covered performance testing a bit low stress testing, but security testing is a very beautiful, theoretically rich area which we did not cover at all. When there is something called reliability analysis which I told you when we did non-functional testing. Typically reliability analysis for both software and hardware work with models that involve probability, like discrete Markov decision processes, hidden Markov models and they need a good amount of statistics and probability to be able to determine the reliability, right which talks about measuring parameters like mean time between failures average down time average up time and so on.

We did not do any of those because those are quite involved theoretically and algorithmically and need you to pick up that kind of maths. Another big area in testing which we have all together completely left out is usability testing. Usability engineering itself is a big area inside that testing for usability is very important, catching up a lot because of all the apps and games and other interfaces that we are dependent on to use our software. Specific part of usability testing is testing of graphics user interface which I did list as a part of the contents of the course, but I did not do because of lack of time. I wanted to, there is a particular style of techniques by which you work with graph models that look a lot like finite state automata using which you can test graphics user interface which I did not do.



Testing of embedded software is very critical because typically embedded software is a safety critical system and it goes through very rigorous testing process. So, I should say that all the coverage criteria based testing that we did so far can very well be used to test embedded software for GUI applications and so on. But specifically for things like embedded software people do what is called real time testing also because such software come with lot of real time constraints like interrupts, timeouts and exception handling features. So, there is specifically a lot of testing done for these kind of software which we did not look at. But what we looked at can be applied to test embedded software and can also be applied to tests GUIs also. Tools I had introduced you to JUnit mainly because the programs that we dealt with they are all compatible with JUnit. It is a nice open source testing tool for Java thing, but apart from JUnit we did not do any other tools.

I will do some tools now as a part of this lecture, give an overview of several other tools and the other thing that we did not do, if you go back to the V model, here after system testing comes acceptance testing, which is a software is released into market and users basically check if a software is working or not. There is not much algorithmic content in that thing typically, you know for an acceptance testing like a car, what they do is to see if the brakes of the automatic braking system is working fine, they try to crash a car and check the brake system. Similarly for the software that controls how the airbags are handled they try to create a scenario to check if the car controller is bringing up the airbags properly or not. So, these things are either done using exhaustive simulation setups or in the real life. So, there is not much algorithmic, mathematical content that can be taught as a part of such a course.

(Refer Slide Time: 22:33)

Why we didn't look at tools?

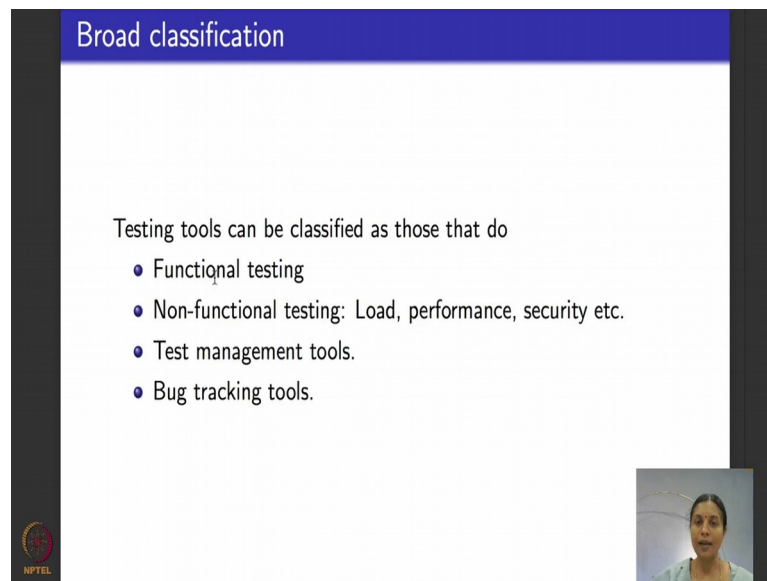
- Tools help in automation of testing.
- Automation is possible only for execution, documentation and maybe, analysis.
- Test case design is to be manually done.
- Learning a tool is basically learning its test interface, notations etc., can be picked up easily.



So, we did not do acceptance testing because of that reason. So, now, for the rest of this lecture, I want to spend some more times on looking at tools. Before we do that I want to justify and tell you why we did not look at tools at all throughout this course. It might be a very natural question that you have. Tools are needed for testing they are indispensable in fact for testing; testing cannot exist without automation with the help of tools. Automation is necessary only for executing a test case, for documenting a test case and maybe use the results for analyzing the test case to see if there is an error in the software or not. But who is going to design the test cases? That has to be done by the tester or a programmer that is what this course focused on algorithms and techniques, for test case design. That is manually done, once that is done, you should be able to use any tool for automation and to be able to execute it.

Also, the other reason why I did not really teach tools is basically what does teaching a tool mean? You download the tool; you learn its interface you learn the commands of the tool or if it has a web interface, you learn the web interface or the IDE. So, basically all our focus will go on figuring out the then trying to learn its interface and how it interacts with the user, yeah, I am sure you would agree with me that that can be picked up on your own and you really do not need a teacher to be able to teach you that. So, typically in a classroom course, we do not focus on tools. It is only if you have a lab component attached to the classroom course that you would do the tool and that also you sort of try to pick up on your own. We typically do not teach it as a part of the routine lectures.

(Refer Slide Time: 24:13)



Broad classification

Testing tools can be classified as those that do

- Functional testing
- Non-functional testing: Load, performance, security etc.
- Test management tools.
- Bug tracking tools.

The slide features a blue header with the title 'Broad classification'. Below the title, the text 'Testing tools can be classified as those that do' is followed by a bulleted list of four categories: Functional testing, Non-functional testing (with sub-points: Load, performance, security etc.), Test management tools, and Bug tracking tools. In the bottom right corner, there is a small video inset showing a woman speaking. The bottom left corner contains the NPTEL logo.

So, what I will do for the rest of this course, this lecture, is to give you an overview of tools. So, tools that involved in testing can be broadly classified into the following categories: tools do functional testing automation, tools that do nonfunctional testing typically, load testing, performance testing, security, vulnerability analysis and tools that are used for test management which is basically to archive, record a test case, to track the progress of testing, to plan your testing, project managers who focus on testing project managers use these tools.

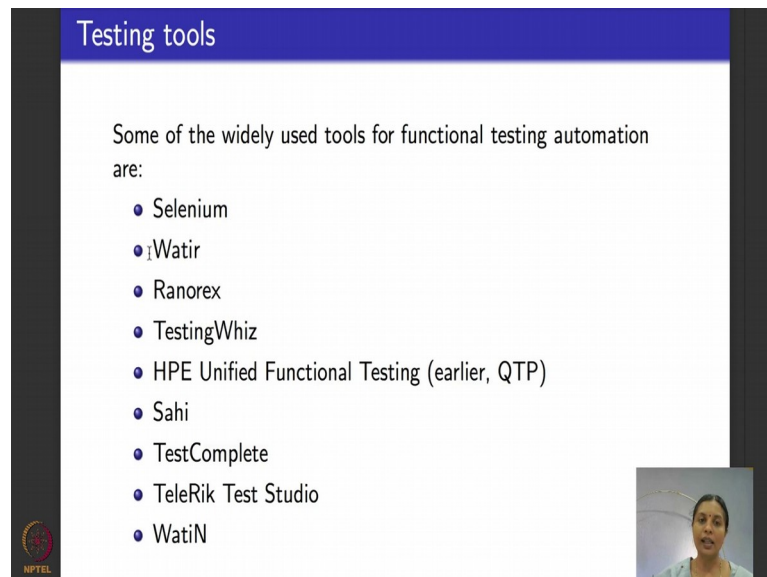
Another important category of tools are what are called bug tracking tools. You might have heard of tools like Jira and all that. Once you do a defect, right, find that effect using testing the idea is to be able to record that defect take it back to the developers see if the developer is able to fix that bug or defect and re-test it again to see if the defect has been mitigated. So, once you find the bug or a defect it needs to be tracked from the time it was found to the time that it is rectified.

So, there are specific tools that help you to track these bugs or defects, those are called bug tracking tools. As a part of my modules on various non-functional testing, I had given you some of the nonfunctional testing tools that are popular and please remember, when we did that web testing, I had given you the most exhaustive web page that I am aware of that contains several different non-functional testing tools. So, what I will do now and I also give you tools for regression testing and mobile apps testing that I am familiar with. What I will do now is I will try to focus on testing tools that will do

functional testing. Some of them will also test for load and performance we will see them as we go on.

Here is what is believed to be the most widely used set of tools for test automation Selenium, Watir, this is pronounced water, tool called Ranorex which I already introduced you this week in another module, testing whiz then there is something called HPEUFT; this tool was recently renamed it was popularly known as QTP, you might have heard about it. There is another tool called Sahi, Sahi and Selenium are used by many companies test complete, Telerik test studio WatiN which was motivated by Wttr.

(Refer Slide Time: 26:22)



The slide is titled "Testing tools" in a blue header. Below the header, it says "Some of the widely used tools for functional testing automation are:". A bulleted list follows, listing the tools: Selenium, Watir, Ranorex, TestingWhiz, HPE Unified Functional Testing (earlier, QTP), Sahi, TestComplete, Telerik Test Studio, and WatiN. In the bottom right corner of the slide, there is a small video inset showing a woman speaking. The NPTEL logo is visible in the bottom left corner of the slide.

Testing tools

Some of the widely used tools for functional testing automation are:



- Selenium
- Watir
- Ranorex
- TestingWhiz
- HPE Unified Functional Testing (earlier, QTP)
- Sahi
- TestComplete
- Telerik Test Studio
- WatiN

So, well just look at a few of these tools and see what they can do.

(Refer Slide Time: 26:41)

Selenium

- Selenium is a suite of tools to automate web browsers across many platforms.
- Selenium
 - runs in many browsers and operating systems,
 - can be controlled by many programming languages and testing frameworks.
- It offers record and playback features to write tests without learning Selenium IDE.
- One of the most used tools.



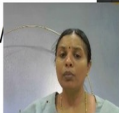

So, selenium is basically a suite of tools to automate web browsers, or web browser based test automation. So, it automates web browsers across several different platforms, Windows based, Linux based and so on. Selenium as I told you runs on many different browsers many different operating systems can be controlled by several programming languages and testing frameworks. It basically gives you a web interface to handle and automate your test execution.

It records; it offers record and playback features which I told you in the context of regression testing, to write tests without learning the IDE and it is considered to be one of the most used tools in the industry.

(Refer Slide Time: 27:24)

Watir

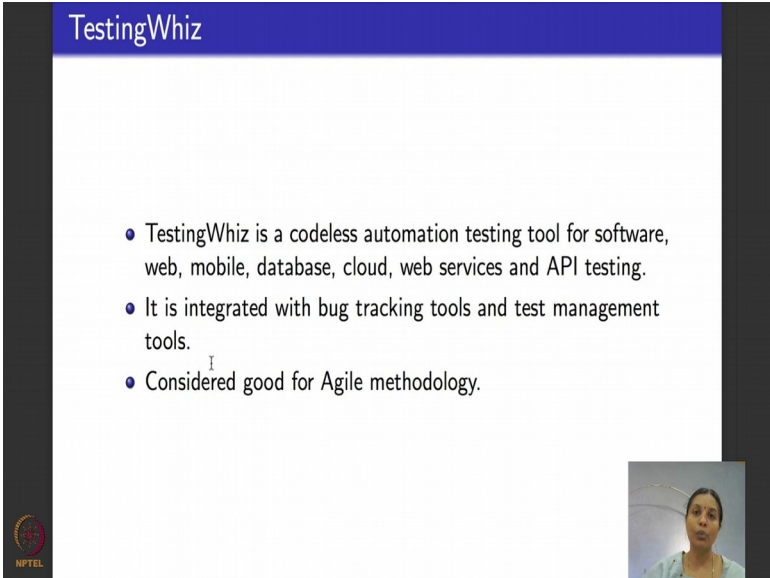
- Watir: Web Application Testing In Ruby, pronounced “water”.
- An open-source family of Ruby libraries for automating web browsers.
- A Watir project consists of smaller projects. Some important ones are:
 - Watir-classic: The Internet Explorer process is the server and serves the automation objects, exposing their methods; while the Ruby program then becomes the client which manipulates the automation objects.
 - Watir-webdriver: A modern version of Watir API based on Selenium.
 - Watirspec: Watirspec is executable specification of the Watir API.



Then there is a tool called Watir, pronounce it as water, it stands for web application testing in Ruby. Ruby is this programming language. It basically is an open source family of libraries that are written in Ruby again for web browser automation, for automating web browser automation and text execution through web browser automation. A Watir project typically consists of smaller pieces Watir classic is the IE process where the server is the IE process and what does it serve, the clients, they are the objects that you want to automate and execute as a part of your test case execution. How does it do? It exposes their methods and then the ruby program in which Watir is written becomes the client then it manipulates the automation objects.

So, that is how the classical version works. There is a web driver Watir which is a modern version of Watir API based on Selenium tool. So, Selenium can be integrated with Watir then there is something called Watir spec, which is basically an executable version of the Watir API.

(Refer Slide Time: 28:28)



TestingWhiz

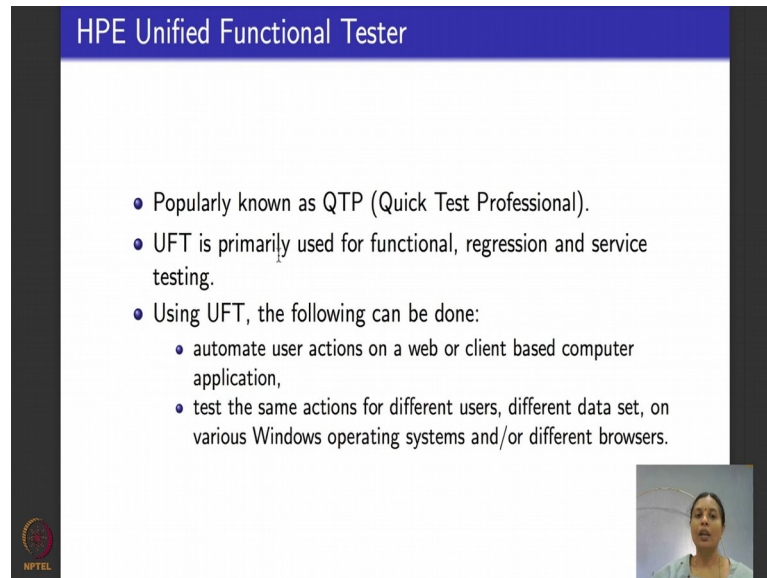
- TestingWhiz is a codeless automation testing tool for software, web, mobile, database, cloud, web services and API testing.
- It is integrated with bug tracking tools and test management tools.
- Considered good for Agile methodology.

MPTEL

Then another popular tool is testing whiz it is a cordless automation testing tool for software testing web applications testing, mobile apps testing; testing for databases like SQL cloud testing; testing by hosting on the cloud, web services which form a specific part of web apps and for API testing. The nice thing about testing whiz is that its integrated with several different bug tracking and test management tools. So, you could actually use the same tool to be able to track your defect, if you have found any and

manage it later, instead of switching over to a separate bug tracking tool and because of this, testing whiz, the belief is that is considered good for people who follow agile methodologies.

(Refer Slide Time: 29:16)



The slide is titled "HPE Unified Functional Tester" in a blue header. It contains a bulleted list of points about the tool. In the bottom right corner, there is a small video inset showing a woman speaking. The NPTEL logo is in the bottom left corner of the slide area.

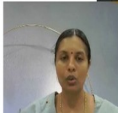

- Popularly known as QTP (Quick Test Professional).
- UFT is primarily used for functional, regression and service testing.
- Using UFT, the following can be done:
 - automate user actions on a web or client based computer application,
 - test the same actions for different users, different data set, on various Windows operating systems and/or different browsers.

The next 1 is QTP; earlier it was known as QTP short form for quick test professional, but now it is called HPE unified function tester. It is primarily used for functional testing regression testing and testing of service apps. You can automate user actions on a web browser. This what always happens in most of the automation tools or you can automate it on a client based computer application that you write for it.

(Refer Slide Time: 29:52)

Sahi

- An open-source test automation tool to automate web applications testing.
- Sahi provides following features:
 - Performs multi-browser testing
 - Supports ExtJS, ZK, Dojo, YUI, etc. frameworks
 - Record and playback on browser testing

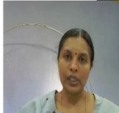



You can test the same actions for different users, for different datasets on various Windows operating systems and various browsers. The next popular tool in our list is Sahi. It is, the nice thing about it is that it is an open source tool. Again for test automation, it automates web application testing. You can use it with several different browsers, it supports all these frameworks that are listed here and it also does record and playback like most other automation tools.

(Refer Slide Time: 30:12)

Telerik TestStudio

- Telerik TestStudio is another tool to automate desktop, web and mobile application testing.
- Includes UI, load, and performance testing.
- Offers various compatibilities like:
 - Support of programming languages like HTML, AJAX, ASP.NET, JavaScript, Silverlight, WPF, and MVC.
 - Record and playback
 - Cross-browser testing
 - Manual testing
 - Integration with bug tracking tools



Telerik Studio is the next in our list. This again automates desktop web and can be used for mobile applications testing. I do not think I listed it in my mobile applications lecture, but I am not sure. This also does nonfunctional testing, it can test for graphical user

interface, load, stress and performance. It offers, supports many programming languages like HTML, AJAX, ASP dot NET, JavaScript and so on. It offers record and playback features. In fact, most of these tools offers record and playback features, supports several browsers, can do manual testing and it is integrated with bug tracking tools. So, this was an overview. I have did not give you URLs because I realized that they keep changing and once I give a URL for such a long list, it is going to be difficult for me to keep track of when the website gets updated, but feel free to Google for them, you will be able to get the URLs well.

So, this will give you a basic set of test tools open source and proprietary ones that are available for free trials for you to start your testing exercise, if you are interested. If you want to call yourself as an expert, complete expert in software testing, apart from this course, you could also learn one of the popular tools like Selenium or Sahi or one of those and the get to familiarize yourself with their interfaces and what their capabilities are and how to do automation of test case execution in them. That will definitely help you if you are interested specifically in job interviews and so on to be able to get a job as a quality expert.

So, I hope you enjoyed this course and we hope these set of lectures were useful for you and you learnt a bit in software testing all the best for your upcoming exams hope to see you all again in NPTEL sometime.

Thank you.

*THIS BOOK IS NOT FOR SALE
NOR COMMERCIAL USE*

PH: (044) 2257 5905(08)

nptel.ac.in

swayam.gov.in